Advances in Equational Theorem Proving – Architecture, Algorithms, and Redundancy Avoidance

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation
von

Dipl.-Inform. Bernd Löchner

Datum der wissenschaftlichen Aussprache: 20. Juli 2005

Dekan: Prof. Dr. Reinhard Gotzhein

Prüfungskommision:
Vorsitzender: Prof. Dr. Arnd Poetzsch-Heffter
Berichterstatter: Prof. Dr. Jürgen Avenhaus
Prof. Dr. Klaus E. Madlener

D 386

Befiehl Du Deine Wege und was Dein Herze kränkt,
der allertreusten Pflege des, der den Himmel lenkt.
Der Wolken, Luft und Winden gibt Wege, Lauf und Bahn,
der wird auch Wege finden, da Dein Fuß gehen kann.

Ihn, ihn laß tun und walten, er ist ein weiser Fürst
und wird sich so verhalten, daß Du Dich wundern wirst,
wenn er, wie ihm gebühret, mit wunderbarem Rat
das Werk hinausgeführet, das Dich bekümmert hat.

Paul Gerhard

Zusammenfassung

Automatisches Theorembeweisen ist ein spezielles Suchproblem, das aufgrund seiner Unentscheidbarkeit sehr schwierig ist. Die Herausforderung bei der Entwicklung praktisch leistungsfähiger Beweiser liegt darin, die reich entwickelte Theorie so in ein Programm abzubilden, daß dieses effizient auf dem Rechner läuft. Ausgehend von einem Schichtenmodell für automatische Theorembeweiser stellen wir in dieser Arbeit verschiedene Techniken vor, die für die Entwicklung leistungsfähiger Gleichheitsbeweiser von Bedeutung sind. Die Beiträge lassen sich in drei Gruppen einteilen.

Architektur. Wir stellen eine neuartige Beweiserarchitektur vor, die eine mengenbasierte Kompressionsstrategie verfolgt. Mit wenig zusätzlichem Berechnungsaufwand läßt sich der Speicherbedarf erheblich reduzieren. Die Architektur hat eine Reihe weiterer schöner Eigenschaften. So können ohne Zusatzaufwand detaillierte Beweisobjekte ausgegeben werden. Weiterhin eröffnet sich der Weg zu einer effizienten Parallelisierung des Beweisers, die in der Praxis sehr gute Ergebnisse zeigt. Schließlich ermöglicht die kompakte Darstellung neue Anwendungen für automatische Gleichheitsbeweiser im Bereich von Verifikationssystemen.

Algorithmen. Um die Geschwindigkeit eines Beweisers zu steigern, gilt es, effiziente Lösungen für die Routinen zu finden, die den Hauptteil der Rechenzeit benötigen. Wir zeigen Beschleunigungen von mehreren Größenordnungen für zwei der am meisten verwendeten Termordnungen, LPO und KBO. Besonders hervorzuheben sind ein neuer generischer Unerfüllbarkeitstest für Ordnungsconstraints, der polynomialen Laufzeitbedarf hat, und darauf aufbauend ein hinreichender Grundreduzierbarkeitstest mit einem hervorragenden Kosten-Nutzen-Verhältnis.

Redundanzvermeidung. Auf Kalkülebene ist der Begriff der Redundanz zentral für die Rechtfertigung simplifizierender Inferenzen. Deren Verwendung hat eine wesentliche Beschränkung des Suchraumes zur Folge. In unserer praktischen Erfahrung ist der Redundanzbegriff der Vervollständigung ohne Abbruch zu schwach. Bei Vorliegen von Assoziativität und Kommutativität werden sehr viele ähnliche Gleichungen erzeugt. Durch Erweiterung und Verfeinerung der Beweisordnung können deutlich mehr Gleichungen als redundant nachgewiesen werden. Außerdem erlaubt der verfeinerte Kalkül, redundante Gleichungen zur Simplifikation zu verwenden, ohne sie für generierende Inferenzen berücksichtigen zu müssen. Wir beschreiben die effiziente Implementierung einiger Redundanzkriterien und untersuchen experimentell ihren Einfluß auf die Beweissuche.

Die Kombination der beschriebenen Techniken führt zu einer erheblichen Steigerung der Leistungsfähigkeit eines Beweisers, wie wir anhand des automatischen Beweisers Waldmeister in ausführlichen Experimenten demonstrieren. Diese Fortschritte erlauben das Lösen von Problemen, die vorher jenseits der Möglichkeiten des Beweisers waren. Mit Hilfe der vorgestellten Techniken lassen sich neue Anwendungsfelder für automatische Gleichheitsbeweiser erschließen.

Summary

Automated theorem proving is a search problem and, by its undecidability, a very difficult one. The challenge in the development of a practically successful prover is the mapping of the extensively developed theory into a program that runs efficiently on a computer. Starting from a level-based system model for automated theorem provers, in this work we present different techniques that are important for the development of powerful equational theorem provers. The contributions can be divided into three areas:

Architecture. We present a novel prover architecture that is based on a set-based compression scheme. With moderate additional computational costs we achieve a substantial reduction of the memory requirements. Further wins are architectural clarity, the easy provision of proof objects, and a new way to parallelize a prover which shows respectable speed-ups in practice. The compact representation paves the way to new applications of automated equational provers in the area of verification systems.

Algorithms. To improve the speed of a prover we need efficient solutions for the most time-consuming sub-tasks. We demonstrate improvements of several orders of magnitude for two of the most widely used term orderings, LPO and KBO. Other important contributions are a novel generic unsatisfiability test for ordering constraints and, based on that, a sufficient ground reducibility criterion with an excellent cost-benefit ratio.

Redundancy avoidance. The notion of redundancy is of central importance to justify simplifying inferences which are used to prune the search space. In our experience with unfailing completion, the usual notion of redundancy is not strong enough. In the presence of associativity and commutativity, the provers often get stuck enumerating equations that are permutations of each other. By extending and refining the proof ordering, many more equations can be shown redundant. Furthermore, our refinement of the unfailing completion approach allows us to use redundant equations for simplification without the need to consider them for generating inferences. We describe the efficient implementation of several redundancy criteria and experimentally investigate their influence on the proof search.

The combination of these techniques results in a considerable improvement of the practical performance of a prover, which we demonstrate with extensive experiments for the automated theorem prover WALDMEISTER. The progress achieved allows the prover to solve problems that were previously out of reach. This considerably enhances the potential of the prover and opens up the way for new applications.

Danksagung

Mein Dank gilt all denen, die auf vielfältige Weise zum Gelingen meiner Promotion beigetragen haben.

Herrn Prof. Dr. J. Avenhaus, der mir die Möglichkeit eröffnet hat, diese Arbeit zu erstellen, danke ich für die langjährige Unterstützung und die vielen Freiräume, die er mir gegeben hat. Herrn Prof. Dr. K. Madlener danke ich für die Übernahme der Zweitbegutachtung meiner Arbeit und die vielen interessanten Diskussionen in den letzten Jahren. Herr Prof. Dr. A. Poetzsch-Heffter hat freundlicherweise die Leitung der Promotionskommision übernommen.

Weiterhin möchte ich mich bei den Kollegen der Arbeitsgruppen von Prof. Avenhaus und Prof. Madlener für alle Zusammenarbeit und Diskussion bedanken. In den ersten Jahren haben mich insbesondere Birgit Reinert, Roland Vogt und Claus-Peter Wirth in vielen Gesprächen begleitet. In den letzten Jahren haben Tobias Schmidt-Samoa und Bernd Strieder mich in vielen Dingen tatkräftig unterstützt.

Waldmeister ist nicht vorstellbar ohne die außerordentlichen Leistungen von Arnim Buch, Thomas Hillenbrand und Andreas Jaeger. Vielen Dank für diesen hervorragenden Grundstock meiner praktischen Arbeiten. Die ausgezeichnete Unterstützung durch die Studenten Jean-Marie Gaillourdet, René Rondot, Christian Schmidt, Hendrik Spies und Thomas Türk hat mir gerade in praktischen Dingen erheblich weitergeholfen.

Die jährlichen Beweiserwettbewerbe und das internationale Zusammentreffen mit an der Implementierung von Beweissystemen interessierten Kollegen waren eine stete Herausforderung und eine gute Inspirationsquelle. Mein Dank gilt Geoff Sutcliffe für die Organisation und das Design der T-Shirts, sowie Peter Baumgartner, Bill McCune, Bernhard Gramlich, Reinhold Letz, Christopher Lynch, Hans de Neville und Christoph Weidenbach für viele inspirierende Diskusssionen. Insbesondere danke ich Thomas Hillenbrand und Stephan Schulz für die langjährige gute Zusammenarbeit und Freundschaft.

Allen, die Teile der Arbeit Korrektur gelesen haben, möchte ich für die vielfältigen Kommentare und die wertvollen Anmerkungen danken. Many thanks to Ursula Schmidt for polishing my (funny) english, for all motivation, and for all prayers.

Mein besonderer Dank gilt meiner Familie für die langjährige Unterstützung und Ermutigung. Meiner Frau Sabine danke ich sehr herzlich für alles Tragen, alle Aufmunterung und alle Geduld.

Contents

1 Introduction **1**

2 Basic notions and notations **7**

3 Logical Foundations **13**
3.1 Completion techniques for equational theorem proving 13
3.2 The inference system $\mathcal{G}$ 18
3.3 Extensions and related work 28

4 Orderings and their efficient implementation **31**
4.1 Preliminaries for the implementation 32
4.2 The Lexicographic Path Ordering 33
4.3 The Knuth-Bendix Ordering 53
4.4 Related work and implementation status 72
4.5 Conclusions 74

5 Ordering constraints **77**
5.1 Syntax and satisfiability of ordering constraints 78
5.2 A decision procedure for the satisfiability of LPO constraints 80
5.3 Sufficient tests for the unsatisfiability of KBO constraints 92
5.4 A generic test for ground reduction orderings 98
5.4.1 Extensions for particular orderings 104
5.5 Experimental evaluation 107
5.6 Conclusions 110

6 Redundancy criteria **113**
6.1 Practical aspects of redundancy criteria 113
6.2 Ground convergent subsystems 115
6.3 Case splitting with variable constraints 118
6.4 Confluence trees 124
6.5 Ground reducibility 137
6.5.1 The AC-case 140
6.5.2 A refined criterion for the AC-case 142
6.6 Experimental evaluation 145
6.7 Related work and concluding remarks 157

7 A prover architecture 159
7.1 An inspection of the proof procedure . . . 159
7.2 The Waldmeister loop . . . 166
7.2.1 A space-efficient representation of $\mathfrak{P}$ . . . 167
7.2.2 The set of active facts $\mathfrak{A}$ and its collected history . . . 173
7.2.3 The new role of $\mathfrak{P}$ . . . 176
7.3 An experimental evaluation of the Waldmeister loop . . . 177
7.4 Further benefits . . . 183
7.5 Related work and concluding remarks . . . 189

8 Conclusions 193

Bibliography 195

1 Introduction

Theorem proving is the task of finding a proof for a conjecture from a set of axioms. Automated theorem proving can be considered as a *search problem*. As such, it can be described by a *search state* (a collection of data) and a set of *search operations* that modify the search state. Together, this comprises the *search space*, a graph with search states as vertices and search operations as edges. The *branching factor* of a search state gives the number of possible search operations for this search state (i. e., the out-degree of the vertex). Starting from an initial state, a *search procedure* explores the search space, until it finds a *solution*, a search state that satisfies a given property. At each search state with branching factor greater one the search procedure has to make a *choice*; such a search state is then a *choice point*. Determining optimal choices is either impossible or at least infeasible, hence search decisions are usually made with some *heuristics*.

In the case of (first-order) unit equational theorem proving, the search state is typically modeled as a set of (universally quantified) *equations*, the search operations are determined by some logical calculus, usually given as an *inference system*, and a search state is a solution if it is trivial to give a proof for the conjecture. Our aim is to establish an efficient automated theorem prover which finds as quickly as possible a proof for a given conjecture. Of course, this is by far nontrivial and cannot always be achieved, as the task is a semi-decidable problem which, in general, results in an infinite search space. It has been observed very early that the search space for automated theorem proving is a very dense graph. Thus, the search procedure has to make many choices, and with each step in the search the branching factor increases considerably. This effect is called *combinatorial explosion*. Automated theorem proving turned out to be a lot harder than originally expected.

Nevertheless, the field has made enormous progress during the last decades, and automated theorem provers nowadays can solve within seconds problems that were considered as very challenging only ten years ago. Hence, automated theorem provers have been successfully used for many different applications. McCune solved the so-called Robbins problem with his automated theorem prover EQP, a mathematical question that withstood human proof attempts for more than sixty years [McC97b]. This achievement made it to the front page of the New York Times. Wolfram uses the prover WALDMEISTER to study single-axiom formulations of Boolean logic [Wol02]. Bos takes SPASS and other automated provers to solve problems in the context of natural language understanding [BB03]. Other applications are the analysis of cryptographic protocols [Wei99], the certification of aerospace code [DFS04], the synthesis of software from components [BRLP98], and many others.

Advances in several areas have contributed to this progress. For the equational case calculi based on completion and rewriting techniques [KB70, BDP89, BG94] have

shown their superiority to previous approaches. Their use of term orderings helps to break the symmetric nature of the equality predicate, therefore restricting the search space considerably. The development of special purpose algorithms and data structures, most notably indexing techniques [McC92], accelerates the basic operations of a prover. The automation of the selection of search heuristics [HJL99, Sch00] considerably improves the practical usability of a prover. In our experience, with their help the prover automatically chooses better heuristics than an experienced user does in a first attempt. Processor speed and main memory of available computers have increased dramatically over the last decades. Modern operating systems and compilers also have improved, but to a lesser extent [Pro98]. Most current high-performance theorem provers are still implemented in C or C++, allowing full utilization of the power of the hardware, but at the cost of a more complex development.

A good *prover architecture* turns out to be invaluable. The formulation of the logical calculi offers us huge freedom in design. The prover architecture closes this design gap. It describes how the search is organized, it defines what the main data structures of the prover are, it specifies how the inferences are aggregated into procedures, it determines which decisions are influenced by heuristics. Thus, it constitutes the central link between the progression of the search on the logical side and the control of the search on the heuristic side.

It is important to distinguish the prover architecture from the *system architecture*. As depicted in Figure 1.1, the prover architecture is one important part of the system architecture. However, as should be clear from the discussion so far, the other levels are important as well; and for each level we have described essential achievements which only together explain the overall progress of the systems. Unfortunately, the levels are not as independent from each other as we would wish. Design decisions in one level affect other levels as well. Many attempts to improve automated theorem proving have focused on one level of the system with sometimes disastrous effects on other levels. As McCune points out [McC97a], the literature contains statements that claim the efficiency of some inference rule because it prunes the search space – without any experimental evaluation. This misses the dynamic nature of the problem. An equation that is unnecessary for theoretical reasons can nevertheless be important for the proof search. It may help to cut down the search space by showing many other equations redundant in a cheap way. Therefore, experiments are an important means to assess the costs and benefits of an approach.[1] Sometimes only experiments can tell us whether something that is theoretically optimal is also useful in a practical context.

The development of a high-performance theorem prover is by far a nontrivial task. Usual methods of software engineering are not directly applicable. The challenge of the development is not to organize hundreds of implementors for a huge project. The challenge is to bring together the rich theoretical background with the practical realities. Take as an example the memory hierarchy of modern machines [HP03]. To get maximal profit from the processor speeds the design and implementation of the algorithms should be *cache aware*. Some people have dismissed the task of mapping the theoretical concepts to programs that run efficiently on a concrete machine as an

[1] The emerging field of experimental algorithmics (see e. g. [FMS02]) shows that in other areas of computer science the importance of experimentation is noticed as well.

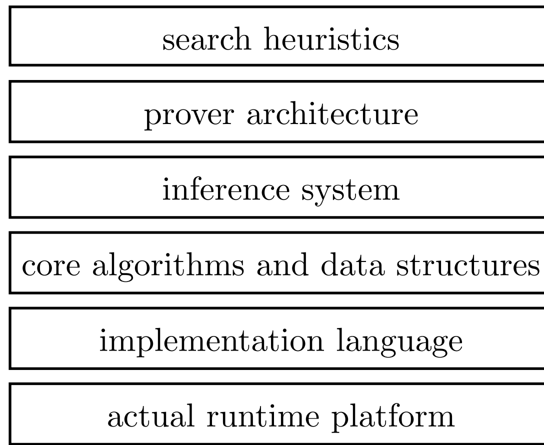


Figure 1.1: A level-based view of the system architecture

engineering issue. We think that it is more appropriate to see it as the central problem of the system development.

This work covers improvements of several levels of the system architecture. As explained, the prover *architecture* is of central importance for an efficient system. Saturating provers have to cope with huge numbers of formulas which makes the memory requirements of such provers a limiting factor. By using a novel set-based compression approach we can dramatically reduce these memory requirements at small additional computational costs. The architectural refinement leads to other advantages as well, for example proof objects can be constructed without further overhead. In addition it turns out that the revealed structure of the search state paves the way to an easily implemented parallelization of a prover with modest communication requirements and rewarding speed-ups.

Algorithms form the core of a theorem prover. With indexing techniques available, new bottlenecks emerge in the systems. Orderings can be responsible for a good deal of the prover's running time. Hence one focus is on the efficient implementation of orderings. For the development process we use an interesting two-level approach. On an abstract level we describe the algorithms with algebraic specification techniques. This allows us to present in a clear way the different optimization steps. Then we translate the different variants into C and evaluate them experimentally. With this two-level approach we gain a clear understanding of the algorithmic problem and well-structured, error-free programs. Other important algorithmic developments concern unsatisfiability tests for ordering constraints and sufficient ground joinability tests.

The *avoidance* of redundancy concerns mainly the inference system level. It is a powerful means to restrict the search space of a prover. For modeling the notion of redundancy as a theoretical concept, constraint-based techniques are state of the art. But these techniques are often non-constructive or at least computationally infeasible, hence they have only rarely made their way into today's provers. In turn, we have recognized in challenging applications of theorem proving that the prover Waldmeister often got stuck enumerating equations that are only permutations of each other; and

in pen-and-paper proofs many of them have turned out to be redundant. This practical experience has motivated our research on deriving feasible constraint-based instances of the concept of redundancy. We have therefore refined the unfailing completion approach of [BDP89]. The chosen notion of redundancy allows us to keep redundant equations for simplification purposes which further helps to prune the search space. Our notion of redundancy is very general and only semi-decidable. We therefore employ several sufficient redundancy criteria which are all based on a strengthened form of ground joinability. Experiments show that with enhanced redundancy elimination the search space is pruned considerably, and demonstrate that the effort we have invested on the theory level pays off in practice.

Most of the work we present has a strong experimental background. Our interest is as well in practical relevance as in theoretical elegance. As it turns out, especially the combination of the developed techniques leads to impressive improvements in the power of a theorem prover and thus considerably advances the state of the art.

Overview of the thesis

The thesis is structured as follows. Chapter 2 contains the basic notions and notations we use throughout the work.

In Chapter 3 we first motivate by several examples why it is advantageous to refine the usual unfailing completion approach. We then present the inference system $\mathcal{G}$ which has an improved support for redundancy elimination. With the refined proof ordering many more equations can be shown redundant. Furthermore, system $\mathcal{G}$ justifies the use of redundant equations for simplifying inferences. As the redundancy of an equation is an undecidable property, we show how a strengthened form of ground joinability can be used as a sufficient indicator.

The next two chapters are devoted to algorithmic problems. In Chapter 4 we develop in a step-by-step manner efficient algorithms for two of the orderings in widespread use (LPO and KBO). Experiments allow us to assess the practical improvements. As such, the chapter is a case study of algebraic specification and program transformation techniques. To our knowledge, the algorithm developed for KBO is the first asymptotically optimal one.

In Chapter 5 we describe ordering constraints which are a powerful means to formulate restrictions for the set of ground instances of a syntactic expression. The most important algorithmic problem with ordering constraints is to decide whether a given constraint is satisfiable. The tests we present are either ordering specific (LPO and KBO) or generic in nature; one test is a decision procedure, the others are sufficient tests. The chapter concludes with an experimental evaluation of these tests concerning accuracy and running time.

With Chapter 6 we come back to redundancy elimination. We develop in detail several criteria which have different trade-offs. Various criteria are quite efficient, but restricted in applicability to certain theories. Others have a more general nature at higher costs. With an extensive experimental evaluation we show that redundancy criteria justify the theoretic and algorithmic apparatus they depend on. Especially for challenging proof tasks they turn out to be very valuable.

A refined prover architecture is the topic of Chapter 7. By using a set-based compression scheme, we achieve a substantial reduction of the memory requirements, at moderate computational costs. Further wins are architectural clarity, the easy provision of proof objects, and a new way to parallelize a prover. As the integration into the WALDMEISTER prover shows, this enhances the potential of a prover considerably.

Finally, Chapter 8 contains a short conclusion and briefly sketches important tasks for future work.

2 Basic notions and notations

We use standard concepts from term rewriting [Ave95, BN98, DP01]. As the main purpose of this chapter is to fix notation we keep it rather concise and omit examples. For a more thorough treatment we refer the reader to the cited literature.

Orderings. A *quasi-ordering* $\gtrsim$ is a reflexive, transitive binary relation. We write $\lesssim$ for its inverse. It defines an equivalence relation by $\approx\, = \,\gtrsim \cap \lesssim$. The *strict part* $> \,=\, \gtrsim - \approx$ is a *partial ordering*, an irreflexive, transitive binary relation. If no confusion can arise we may use the strict part to refer to the quasi-ordering. Sometimes, we define a partial ordering with $=$ as equivalence, which we write as $\geq$. An ordering $>$ is *well-founded* (on A) if there is no infinite descending chain $a_1 > a_2 > a_3 > \ldots$ (with $a_i \in A$). For example, on the naturals $\mathbb{N} = \{0, 1, 2, \ldots\}$ the usual "greater than" ordering $>_{\mathbb{N}}$ is well-founded.

For two quasi-orderings $\gtrsim_1$ and $\gtrsim_2$ the *lexicographic combination* $\gtrsim\, = (\gtrsim_1, \gtrsim_2)$ is defined as $(a, b) \gtrsim (a', b')$ iff $a >_1 a'$ or $a \approx_1 a'$ and $b \gtrsim_2 b'$. The strict part $>$ of the lexicographic combination is well-founded iff the strict parts $>_1$ and $>_2$ are well-founded. The generalization of the lexicographic combination to n quasi-orderings is straightforward: $(\gtrsim_1, \gtrsim_2, \ldots, \gtrsim_n) = (\gtrsim_1, (\gtrsim_2, (\ldots, \gtrsim_n) \cdots))$.

For a set S the set S^* denotes the set of all finite sequences or *words* over S, that is, the empty word $\lambda \in S^*$ and for $a \in S$ and $w \in S^*$ the word $a.w \in S^*$. For a quasi-ordering $\gtrsim$ on S the *lexicographic extension* $\gtrsim^*$ is defined as $w \gtrsim^* \lambda$ for all $w \in S^*$ and $a.w \gtrsim^* a'.w'$ iff $a > a'$ or $a \approx a'$ and $w \gtrsim^* w'$.

A *multiset* over a set S is a mapping $M : S \to \mathbb{N}$ such that $\{a \mid M(a) \neq 0\}$ is finite. Informally, it is a set with repeated occurrences of its elements such as $\{a, a, a, b\}$. The union of two multisets M_1 and M_2 is defined as $M_1 \cup M_2(x) = M_1(x) + M_2(x)$. The difference of two multisets M_1 and M_2 is defined as $M_1 - M_2(x) = \max\{0, M_1(x) - M_2(x)\}$. For a partial ordering $>$ on S the *multiset extension* $\gg$ on multisets over S is defined as $M \gg M'$ iff there are multisets N, N' such that $\varnothing \neq N \subseteq M$, $M' = (M - N) \cup N'$, and for each $a \in N'$ there is a $b \in N$ with $b > a$. The multiset extension $\gg$ is well-founded iff $>$ is well-founded.

Signatures. A signature $sig = (\mathcal{S}, \mathcal{F}, \alpha)$ consists of a set $\mathcal{S}$ of *sorts*, a set $\mathcal{F}$ of *function symbols* (or *operators*) and an *arity function* $\alpha : \mathcal{F} \to \mathcal{S}^+$. Instead of $\alpha(f) = s_1 \ldots s_n s$ for $f \in \mathcal{F}$ we write $f : s_1 \ldots s_n \to s$ to distinguish the *argument sorts* $s_1, \ldots, s_n$ from the *result sort* s. Function symbols with $n = 0$ are *constants*, with $n = 1$ they are *unary operators*, with $n = 2$ they are *binary operators*. We assume that for each sort $s \in \mathcal{S}$ set $\mathcal{F}$ contains at least one constant with result sort s. We use f, g, h as symbols for operators and a, b, c, d for constants. A *precedence* is an ordering on $\mathcal{F}$.

Terms. Let $(\mathcal{V}_s)_{s\in\mathcal{S}}$ be a system of pairwise disjoint, denumerable sets of *variables* and $\mathcal{V} = \bigcup_{s\in\mathcal{S}} \mathcal{V}_s$. We require $\mathcal{V} \cap \mathcal{F} = \varnothing$. The sets $\mathrm{Term}_s(\mathcal{F},\mathcal{V})$ of terms of sort $s \in \mathcal{S}$ over $\mathcal{F}$ and $\mathcal{V}$ are simultaneously defined via (i) $\mathcal{V}_s \subseteq \mathrm{Term}_s(\mathcal{F},\mathcal{V})$ and (ii) for $f \in \mathcal{F}$ with $f : s_1 \ldots s_n \to s$ and $t_i \in \mathrm{Term}_{s_i}(\mathcal{F},\mathcal{V})$ for $i = 1,\ldots,n$ is $f(t_1,\ldots,t_n) \in \mathrm{Term}_s(\mathcal{F},\mathcal{V})$. The set of *terms* $\mathrm{Term}(\mathcal{F},\mathcal{V})$ over $\mathcal{F}$ and $\mathcal{V}$ is then given by $\mathrm{Term}(\mathcal{F},\mathcal{V}) = \bigcup_{s\in\mathcal{S}} \mathrm{Term}_s(\mathcal{F},\mathcal{V})$. As can be seen from this definition, we usually use prefix notation for terms. However, for some binary operators, such as $+$ or $*$, we use infix notation to enhance readability. We use symbols x, y, z for variables, s, t, u, v, w, l, r for terms, and $\equiv$ for syntactic identity of terms.

With $\mathrm{top}(t)$ we denote the top-symbol of term t. That is, $\mathrm{top}(f(t_1,\ldots,t_n)) = f$ and $\mathrm{top}(x) = x$. For a nonvariable term $t \equiv f(t_1,\ldots,t_n)$ the function $\arg(t)$ delivers $t_1,\ldots,t_n$, the *arguments* of f. The *length* $|t|$ of a term t is the number of symbols it contains. Therfore, the function $|\cdot| : \mathrm{Term}(\mathcal{F},\mathcal{V}) \to \mathbb{N}$ is given by $|x| = 1$ and $|f(t_1,\ldots,t_n)| = 1 + |t_1| + \ldots |t_n|$. We write $|t|_x$ for the *x-length* of t, that is the number of occurrences of symbol $x \in \mathcal{F} \cup \mathcal{V}$ in t. The *depth* $\mathrm{depth}(t)$ of a term t is given as follows: $\mathrm{depth}(t) = 1$ if t is a variable or a constant and $\mathrm{depth}(f(t_1,\ldots,t_n)) = 1+\max\{\mathrm{depth}(t_1),\ldots,\mathrm{depth}(t_n)\}$. The function $\mathrm{Var} : \mathrm{Term}(\mathcal{F},\mathcal{V}) \to 2^{\mathcal{V}}$ computes the variables occurring in a term. We use $\mathrm{Var}(T)$ or $\mathrm{Var}(t_1,\ldots,t_n)$ as simplified notation for $\bigcup_{t\in T} \mathrm{Var}(t)$ or $\mathrm{Var}(t_1) \cup \ldots \cup \mathrm{Var}(t_n)$ respectively. We have $\mathrm{Var}(x) = \{x\}$ and $\mathrm{Var}(f(t_1,\ldots,t_n)) = \mathrm{Var}(t_1,\ldots,t_n)$. A term t containing no variable, that is $\mathrm{Var}(t) = \varnothing$, is a *ground term* and $\mathrm{Term}(\mathcal{F}) = \mathrm{Term}(\mathcal{F},\varnothing)$ is the set of all ground terms over $\mathcal{F}$.

Extended signatures. It is sometimes necessary to consider extensions of $\mathcal{F}$. In theorem proving new operators are introduced by Skolemization. Furthermore, some properties are not stable against signature extensions. Conversely, some properties may be formulated for a signature intended to be a proper subset of the actual signature. We write $\mathcal{F}^{\mathrm{e}}$ for some arbitrary, but fixed extension of $\mathcal{F}$. Therefore, we have $\mathrm{Term}(\mathcal{F}) \subsetneq \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ as $\mathcal{F} \subsetneq \mathcal{F}^{\mathrm{e}}$.

Subterms and positions. A *subterm* of a term t is either t itself, or if t is of the form $f(t_1,\ldots,t_n)$ it is a subterm of one of the t_i, $i = 1,\ldots,n$. A *proper subterm* of a term t is a subterm distinct from t. The subterm ordering $\succ_{\mathrm{st}}$ is given by $s \succ_{\mathrm{st}} t$ iff t is a proper subterm of s. The *homeomorphic embedding* $\succ_{\mathrm{emb}}$ generalizes the subterm ordering. It is $s \succcurlyeq_{\mathrm{emb}} x \in \mathcal{V}$ iff $x \in \mathrm{Var}(s)$ and $s \equiv f(s_1,\ldots,s_n) \succcurlyeq_{\mathrm{emb}} g(t_1,\ldots,t_m) \equiv t$ iff $s_i \succcurlyeq_{\mathrm{emb}} t$ for some $i = 1,\ldots,n$ or $f = g$ and $s_j \succcurlyeq_{\mathrm{emb}} t_j$ for all $j = 1,\ldots,n$. A *position* is a finite sequence of positive integers which we use to refer to a specific subterm of a term. We use p, q to denote positions, λ is the empty position. Associated to a term t is the set of all its positions $\mathcal{O}(t)$. We have $\mathcal{O}(x) = \{\lambda\}$ and $\mathcal{O}(f(t_1,\ldots,t_n)) = \{\lambda\} \cup \{i.p \mid i \in \{1,\ldots,n\}, p \in \mathcal{O}(t_i)\}$. For $p \in \mathcal{O}(t)$ we write $t|_p$ for the subterm of t at position p. It is $t|_\lambda \equiv t$ and $f(t_1,\ldots,t_n)|_{i.p} \equiv t_i|_p$. If $t|_p \in \mathcal{V}$ then p is called a *variable position* of t.

Subterm replacement. We write $t[s]_p$ for the result of *replacing* the subterm $t|_p$ by s. Therefore, $t[s]_\lambda \equiv s$ and $f(\ldots t_i \ldots)[s]_{i.p} \equiv f(\ldots t_i[s]_p \ldots)$. The unaffected part of

t is the *context* of the replacement. To indicate that a term t contains a term s as some subterm we write $t[s]$. Occasionally we extend these notions from terms to other syntactic expressions containing terms.

Substitutions. A *substitution* $\sigma : \mathcal{V} \rightarrow \mathrm{Term}(\mathcal{F}, \mathcal{V})$ is a sort-preserving mapping from variables to terms with finite *domain* $\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \not\equiv x\}$. If $x \in \mathrm{dom}(\sigma)$ then x is *bound* by σ and $\sigma(x)$ is the *binding of* x. For $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$ we may write σ in its *binding form* $\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$. The *image* of σ is given by $\mathrm{im}(\sigma) = \{\sigma(x) \mid x \in \mathrm{dom}(\sigma)\}$. With $\mathcal{I}(\sigma) = \mathrm{Var}(\mathrm{im}(\sigma))$ we denote the variables in the image of σ. A substitution σ with $\mathcal{I}(\sigma) = \varnothing$ is a *ground substitution.* As usual, σ is extended to a mapping from terms to terms by $\sigma(f(t_1, \ldots, t_n)) \equiv f(\sigma(t_1), \ldots, \sigma(t_n))$. The composition $\sigma \circ \theta$ of two substitutions σ and θ is given by $\sigma \circ \theta(t) = \sigma(\theta(t))$ for all $t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. We define the relations $\gtrsim$, $\dot\sim$, and $>$ on substitutions as $\sigma \gtrsim \theta$ iff there is some substitution ρ such that $\sigma = \rho \circ \theta$, $\sigma \dot\sim \theta$ iff $\sigma \gtrsim \theta$ and $\theta \gtrsim \sigma$, and $\sigma > \theta$ iff $\sigma \gtrsim \theta$ and not $\sigma \dot\sim \theta$. If $\sigma \dot\sim \theta$ then there is a variable renaming ξ (i. e., a substitution that is a bijection of variables) such that $\xi \circ \sigma = \theta$. Occasionally, we consider these relations with respect to some proper subset $\mathcal{V}_0$ of $\mathcal{V}$.

If t is a term and σ a substitution, the term $s \equiv \sigma(t)$ is an *instance* of t and if there is no θ such that $t \equiv \theta(s)$ it is a *proper instance.* The *subsumption ordering* on terms $\succcurlyeq_{\mathrm{sub}}$ is given as $s \succcurlyeq_{\mathrm{sub}} t$ iff s is an instance of t. The *encompassment ordering* $\trianglerighteq$ combines the notions of instance and subterm: We have $s \trianglerighteq t$ iff there is some substitution σ such that $s \succcurlyeq_{\mathrm{st}} \sigma(t)$. If $s \trianglerighteq t$ and $t \trianglerighteq s$ the terms s and t are variable renamings of each other which we write as $s \triangleq t$. With $\triangleright$ we denote the strict part of $\trianglerighteq$. If $s \equiv \sigma(t)$ is a ground term s is a *ground instance* of t and σ is called a *grounding substitution.* We write $\mathrm{GSub}(t_1, \ldots, t_n)$ for the set of all grounding substitutions of terms $t_1, \ldots, t_n$ mapping into $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$. If E is some arbitrary syntactic expression containing the terms $t_1, \ldots, t_n$ we use occasionally $\mathrm{GSub}(E)$ as shorthand notation for $\mathrm{GSub}(t_1, \ldots, t_n)$.

Matching and unification. Let $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. If $t \equiv \sigma(s)$ the substitution σ is called a *match* from s to t. If one exists, the match is uniquely determined on $\mathrm{Var}(s)$. We write $\sigma = \mathrm{match}(s, t)$ to denote the match from s to t. A *unifier* of s and t is a substitution such that $\sigma(s) \equiv \sigma(t)$. The substitution σ is called a *most general unifier (mgu)* of s and t iff for any unifier θ of s and t relation $\theta \gtrsim \sigma$ holds (with respect to $\mathrm{Var}(s, t)$). Hence, the mgu is uniquely determined only up to a variable renaming. We write $\sigma = \mathrm{mgu}(s, t)$ to denote an mgu of s and t if one exists. We assume that an mgu σ is always idempotent, i. e., $\sigma \circ \sigma = \sigma$.

Rewrite relations and reduction orderings. Let $\longrightarrow$ be a binary relation. We write $\longleftarrow$ for its inverse, $\longleftrightarrow$ for its symmetric closure, $\xrightarrow{+}$ for its transitive closure, $\xrightarrow{*}$ for its reflexive-transitive closure, and $\xleftrightarrow{*}$ for its symmetric-reflexive-transitive closure. The composition $\longrightarrow_1 \circ \longrightarrow_2$ of two relations $\longrightarrow_1$ and $\longrightarrow_2$ is given for s and t as $s \longrightarrow_1 \circ \longrightarrow_2 t$ whenever there is some u such that $s \longrightarrow_1 u$ and $u \longrightarrow_2 t$. For relation $\longrightarrow$ the *joinability relation* $\downarrow$ is defined by $\downarrow = \xrightarrow{*} \circ \xleftarrow{*}$. The relation $\longrightarrow$ has the

Church-Rosser property iff $\overset{*}{\longleftrightarrow} \subseteq \downarrow$, it is *confluent* iff $\overset{*}{\longleftarrow} \circ \overset{*}{\longrightarrow} \subseteq \downarrow$, and it is *locally confluent* iff $\longleftarrow \circ \longrightarrow \subseteq \downarrow$.

A binary relation $\longrightarrow$ on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ is *stable* against substitutions if $u \longrightarrow v$ implies $\sigma(u) \longrightarrow \sigma(v)$, it is *monotonic* if $u \longrightarrow v$ implies $t[u]_p \longrightarrow t[v]_p$. A *rewrite relation* on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ is a binary relation that is stable and monotonic. An ordering $\succ$ on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ is a *reduction ordering* if it is a rewrite relation and well-founded. It is a *ground reduction ordering* if it is in addition total on $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$. For example, the homeomorphic embedding $\succ_{\mathrm{emb}}$ is a reduction ordering but not a ground reduction ordering – whereas any *lexicographic path ordering* (LPO) or *Knuth-Bendix ordering* (KBO) based on a total precedence on $\mathcal{F}^{\mathrm{e}}$ is a ground reduction ordering.(See Chapter 4 for more information about LPO and KBO). With $c_{\min}$ we denote the $\succ$-smallest constant in $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$. The relation $\longrightarrow$ is *terminating* if $\longrightarrow \subseteq \succ$ for some reduction ordering $\succ$. A terminating and confluent relation is *convergent*. A term t is *reducible* by $\longrightarrow$ iff there is some u such that $t \longrightarrow u$, otherwise it is *irreducible*. We write $s \overset{!}{\longrightarrow} t$ if $s \overset{*}{\longrightarrow} t$ and t is irreducible. Then t is a *normal form* of s. If $\longrightarrow$ is terminating each term has at least one normal form, if it is convergent each term has exactly one normal form. With $s\downarrow$ we denote an arbitrary normal form of s.

Properties on ground terms. For theorem proving several important notions rely on properties of rewrite relations on ground terms. As for example confluence on ground terms is not stable against signature extensions[1] we require $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ as ground term universe in the following definitions. (Recall that substitutions in $\mathrm{GSub}(t_1, \ldots, t_n)$ are mappings from terms in $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ to terms in $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$.) This is appropriate for *deductive* theorem proving, where new function symbols may be introduced by Skolemization. Of course, for *inductive* theorem proving, the situation is different as there the used ground term universe is of great importance for the validity of theorems.

Relation $\longrightarrow$ is *ground confluent* iff $s \overset{*}{\longleftarrow} u \overset{*}{\longrightarrow} t$ implies $s \downarrow t$ for $s, t, u \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$. Relation $\longrightarrow$ is *ground convergent* if $\longrightarrow$ is ground confluent and terminating. Terms $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$ are *ground joinable*, denoted by $s \Downarrow t$, iff $\sigma(s) \downarrow \sigma(t)$ for each $\sigma \in \mathrm{GSub}(s, t)$. A term $t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$ is *ground reducible* iff $\sigma(t)$ is reducible for each $\sigma \in \mathrm{GSub}(t)$.

Equations and rules. An *equation* is of the form $u = v$ with $u, v \in \mathrm{Term}_s(\mathcal{F}, \mathcal{V})$ for some sort $s \in \mathcal{S}$. We write $u \doteq v$ to mean either $u = v$ or $v = u$. We denote by E a set of equations. The *single-step replacement relation* $\longmapsto_E$ is given as $s \longmapsto_E t$ iff there is some $u \doteq v$ in E, $p \in \mathcal{O}(s)$, and substitution σ such that $s|_p \equiv \sigma(u)$ and $t \equiv s[\sigma(v)]_p$. Note that $\longmapsto_E$ is sort-preserving by definition. The relation $=_E \; = \overset{*}{\longmapsto}_E$ is the smallest congruence on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ defined by E.

A *rule* is of the form $l \rightarrow r$ with $l, r \in \mathrm{Term}_s(\mathcal{F}, \mathcal{V})$ for some sort $s \in \mathcal{S}$ and $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$. A set of rules R is a *rewrite system*. The *rewrite relation* $\longrightarrow_R$ is given as $s \longrightarrow_R t$ iff there is some $l \rightarrow r$ in R, $p \in \mathcal{O}(s)$, and substitution σ such that $s|_p \equiv \sigma(l)$ and $t \equiv s[\sigma(r)]_p$. By definition $\longrightarrow_R$ is sort-preserving. The congruence $=_R$ defined by

[1] Consider $\mathcal{F} = \{f, a\}$, $\mathcal{F}^{\mathrm{e}} = \{f, a, b\}$, and $R = \{f(x, y) \rightarrow x, f(x, y) \rightarrow y\}$. The relation $\longrightarrow_R$ is confluent on $\mathrm{Term}(\mathcal{F})$ (the only normal form is a), but not on $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$.

the rewrite system R is given by $=_R \; = \; \overset{*}{\longleftrightarrow}_R$. A rewrite system R is *canonical*, if it is convergent and maximally interreduced. That is, for every rule $l \to r$ in R, term r is irreducible by R and term l is irreducible by $R - \{l \to r\}$.

Ordered rewriting. Given R, E, and a reduction ordering $\succ$, we write $R \subseteq \succ$ if $l \succ r$ for each $l \to r$ in R. Furthermore $E^{\succ} = \{\sigma(u) \to \sigma(v) \mid u \doteq v \text{ in } E, \, \sigma(u) \succ \sigma(v)\}$ is the set of *orientable instances* of equations in E. Let $R(E) = R \cup E^{\succ}$. We call the triple $(R, E, \succ)$ an *ordered rewrite system*. It is terminating, confluent, etc., if the rewrite relation $\longrightarrow_{R(E)}$ is. Rewriting with orientable instances of unoriented equations is called *ordered rewriting*. As we can consider rules as oriented equations we use the term equation if we do not want to distinguish between the two notions.

Proofs. A *proof* in $(R, E, \succ)$ for an equation $s = t$ is a finite sequence of terms $(t_0, \ldots, t_n)$ such that $s \equiv t_0$, $t \equiv t_n$, and for each $i \in \{1, \ldots, n\}$, either $t_{i-1} \longrightarrow_{R(E)} t_i$, or $t_{i-1} \longleftarrow_{R(E)} t_i$, or $t_{i-1} \longmapsto\!\!\dashv_E t_i$. It is a *rewrite proof* if there is some $k \in \{0, \ldots, n\}$, such that $t_{i-1} \longrightarrow_{R(E)} t_i$ for all $i \in \{1, \ldots, k\}$ and $t_{i-1} \longleftarrow_{R(E)} t_i$ for all $i \in \{k+1, \ldots, n\}$.

Rewriting strategies. For given $(R, E, \succ)$ and term t several subterms of t may be candidates for rewrite steps. Such a subterm is called a *redex*. A *rewriting strategy* selects one of the possible redexes. Well-known rewriting strategies are *leftmost-innermost* and *leftmost-outermost*. The first chooses the leftmost redex that contains no other redex, the latter selects the leftmost redex that is not contained in any other redex.

Rewriting with extra variables. For a rewrite step of t at position p with rule $l \to r$ the substitution σ is uniquely determined by a match from l to $t|_p$. For rewrite steps with an equation $u = v$ the following problem may happen: Term v may contain variables that do not occur in u. For such an *extra variable* x the match from u to $t|_p$ does not determine the binding. We are free to extend the match by any binding as long as the ordering restriction $\sigma(u) \succ \sigma(v)$ is fulfilled. In an implementation it is unreasonable to investigate several possible choices. Instead we choose for $\sigma(x)$ the term $c_{\min}$, which seems by its $\succ$-minimality the most "efficient" choice. Therefore, the relation $E^{\succ}$ is of more theoretical interest. With $E_{\succ}$ we denote the relation used in practice, which is given as

$$E_{\succ} = \{\sigma(u) \to \sigma(v) \mid u \doteq v \text{ in } E, \, \sigma(u) \succ \sigma(v), \text{ and } \\ \sigma(x) \equiv c_{\min} \text{ for all } x \in \mathrm{Var}(v) - \mathrm{Var}(u)\} \; .$$

Although $E_{\succ} \subseteq E^{\succ}$, reducibility on $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ is preserved if $\succ$ is a ground reduction ordering. However, for reducibility on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ a more elaborate approach is necessary (cf. [Hil00, Chapter 2.6]).

Overlaps and critical pairs. An *overlap* of equation $u = v$ into equation $s = t$ is defined as follows: Let $p \in \mathcal{O}(s)$ be a nonvariable position in s and $\sigma = \mathrm{mgu}(u, s|_p)$ the most general unifier of u and $s|_p$. Then $\sigma(s)[\sigma(v)]_p = \sigma(t)$ is an overlap if $\sigma(v) \not\succeq \sigma(u)$. It is a *critical pair* if additionally $\sigma(t) \not\succeq \sigma(s)$. Let $\mathrm{OL}(s = t, E)$ be the set of overlaps of elements of E into $s = t$ and let $\mathrm{CP}(R, E)$ denote the set of all critical pairs computable from rules in R and equations in E.

3 Logical Foundations

In this chapter we present the logical foundations our practical work is based on. We are interested in equational theorem proving of the following form: The axiomatization consists of a set E of equations. The conjecture is some single equation $u = v$. All equations are universally quantified. The question is whether $u = v$ is a logical consequence of E. For problems of this kind methods based on term rewriting and completion techniques form the foundation for the most powerful general provers.[1]

In Section 3.1 we motivate by several examples why we use a refinement of the standard unfailing completion approach. Our inference system and the proof of its main properties, such as soundness and completeness, are the topics of Section 3.2. This chapter concludes with a brief discussion of extensions and related work in Section 3.3.

3.1 Completion techniques for equational theorem proving

The input for an equational theorem prover is a *specification* $spec = (sig, E)$, consisting of a signature *sig* and a set of equations E, and a conjecture $u = v$. All equations are implicitly universally quantified. We want to know whether $u = v$ is a logical consequence of E. By Birkhoff's Theorem [Bir35] this is equivalent to the question whether $u =_E v$. This is known as the *word problem* for E.

In the last years rewriting and completion techniques have been shown to be the most powerful tools to solve this problem. The approach is to saturate E into a rewrite system R such that $\bar{u} = \bar{v}$, the Skolemized version of $u = v$, can be proved by joinability. In [KB70] it is shown how to compute such a rewrite system R that is convergent and equivalent to E, that is $=_E \; = \; \overset{*}{\longleftrightarrow}_R$: After selecting a reduction ordering $\succ$ equations are oriented into rules according to $\succ$. Then the representation is processed by adding (simplified) critical pairs that are not already joinable by R. These new equations are then turned into rules and so forth. If this *completion* of E into R terminates successfully we have a convergent rewrite system equivalent to E. This establishes a decision procedure for the word problem for E.

Compared to alternative methods for equational theorem proving, such as paramodulation [RW69], this approach has several benefits: the search space is significantly reduced by $\succ$, only critical pairs, not arbitrary overlaps, need to be considered during the saturation, and furthermore powerful simplification and interreduction mechanisms are available.

[1] For certain domains very powerful systems have been built that are based on domain-specific techniques, such as computer algebra.

There are two main problems with this approach: First, there are specifications for which the word problem is undecidable, hence the completion process cannot always stop successfully. Second, there are equations such as the commutativity axiom $x + y = y + x$ that are inherently unorientable by any reduction ordering $\succ$.

Both problems are overcome (to some extent) by the *unfailing completion* (UKB) approach [HR87, BDP89]. Here, the completion process is interleaved with the joinability test for $\bar{u} = \bar{v}$. From unorientable equations the orientable instances are used for simplification. For the computation of critical pairs, however, the equations are considered in both directions. All in all, we get a semi-decision procedure for the word problem for E.

UKB has some nice properties: If the completion stops and the final system contains no unorientable equation, the final rewrite system is convergent. If furthermore interreduction is performed exhaustively, UKB will find the (up to variable renamings unique) canonical rewrite system for E if it exists. If the final system contains unorientable equations, it is still ground convergent. Therefore, UKB has been the method of choice for (unit) equational logic and today the most powerful theorem provers for this domain implement some variant of UKB.

However, in practice UKB's behavior is sometimes not really satisfactorily, especially when some operators are associative and commutative (AC). With $\mathcal{F}$ we denote the subset of AC-operators. For the following examples we assume that $+ \in \mathcal{F}$ and write $x_1 + \ldots + x_{n-1} + x_n \equiv x_1 + (\ldots + (x_{n-1} + x_n))$ and $nx \equiv x + \ldots + x$ for all $n \in \mathbb{N}$. This notation gives rise to the following concept: For a term $t \equiv t_1 + \ldots + t_n$ the subterms t_i, $i = 1, \ldots, n$, are *AC-subterms* iff $\text{top}(t_i) \neq +$.

Example 3.1 (AC) *Let $\succ$ be some LPO or some KBO. Consider the unfailing completion of the associativity axiom (A) together with the commutativity axiom (C):*

$$(A) \quad (x + y) + z = x + (y + z) \qquad (C) \quad x + y = y + x$$

Standard UKB diverges and produces

$$x_1 + x_2 + \ldots + x_n = x_{\pi(1)} + x_{\pi(2)} + \ldots + x_{\pi(n)}$$

for each $n \geq 2$ and for almost each permutation π. The number of these equations grows exponentially with n.

This indicates the reason why many state-of-the-art provers based on this technique may quickly reach their limit whenever some of the operators are associative and commutative: They get stuck in an overwhelmingly large set of newly generated equations. The commutativity axiom (C) is applicable at each occurrence of $+$ and thus creates many critical pairs. Together with the associativity axiom (A) it generates an infinite set of equations as nicely demonstrated by Example 3.1.

A prominent approach for tackling this problem is to work *modulo AC*, which is well investigated [LB77, PS81, JK86]. The famous proof of the Robbins theorem was found by a prover working modulo AC [McC97b]. But there are several drawbacks: Orderings are restricted to be AC-compatible, matching modulo AC is an NP-complete problem

[BKN87], and AC-unification is very prolific (there may be a doubly exponential number of unifiers for two terms [Dom92, KN92][2]). Furthermore it is very complicated to enhance an existing prover to work modulo AC as core routines and core data structures (e. g. for term indexing) are affected.

We use a different approach, which has the following starting point:

PROPOSITION 3.1 ([MN90]) *Let E be a finite set of equations over sig $= (\mathcal{S}, \mathcal{F}, \alpha)$ and $\succ$ a decidable ground reduction ordering. If $E^{\succ}$ is ground confluent then the word problem for E is decidable.* □

Note that here our notion of ground confluence is important: Because we require confluence on Term($\mathcal{F}^{\mathrm{e}}$) and $\mathcal{F}^{\mathrm{e}}$ is an arbitrary extension of $\mathcal{F}$, the introduction of new function symbols by Skolemization is covered properly. Thus, $u =_E v$ iff $\bar{u} \downarrow \bar{v}$ in $E^{\succ}$ where $\bar{u} = \bar{v}$ is the Skolemized version of $u = v$.

By Proposition 3.1, it is therefore sufficient to saturate a given set E of equations only for ground confluence, which is often easier to achieve than confluence. This allows us to enhance UKB by integrating additional redundancy criteria. These criteria detect equations that are not needed for establishing ground confluence. An equation is redundant if each ground instance has a smaller proof. Because this is an undecidable property we approximate it by ground joinability: Instead of searching arbitrary smaller proofs we consider only smaller rewrite proofs. As ground joinability is still undecidable, we have to use approximation techniques.

In contrast to working modulo a fixed theory we may therefore employ a variety of criteria which offer different cost-benefit ratios. Some of them are cheap to compute, but are limited to fixed theories. Others are generic for a higher cost. It is rather easy to extend an existing prover with such redundancy criteria in comparison to enhancing it to work modulo some theory. The development and testing effort already invested therefore is not lost.

The following examples show that the special treatment of ground joinable equations can impressively reduce the computational effort of the unfailing completion procedure UKB. The examples are rather simple, for easy understanding, but note that in practice these equations often appear as subsystems of more complex specifications.

EXAMPLE 3.2 (AC, CONTINUED) *Consider again the unfailing completion of axioms (A) and (C) when $\succ$ is some LPO or some KBO. Ground joinable equations are deleted. The completion stops with the following system:*

$$\begin{array}{rcl} (x+y)+z & \rightarrow & x+(y+z) \\ x+y & = & y+x \end{array} \qquad (\mathsf{C}')\ x+(y+z) = y+(x+z)$$

Thus, we can show that with one exception all permutations with $n > 2$ are superfluous for ground confluence. We will refer to the found system later with (ACC'). The

[2] The proof of the Robbins theorem was found with an *incomplete* strategy discarding the vast majority of AC-unifiers [McC97b]. For example, with this strategy the prover EQP considers only 139 out of the 1 044 569 AC-unifiers of the unification problem $x + x + x = x_1 + x_2 + x_3 + x_4$!

specialized redundancy test used here is based on AC-equality. Its implementation is very simple and runs in polynomial time (see Section 6.2 for details).

The use of this test leads to speed-up factors of over 20, even when heuristics are used that put permutative equations at a disadvantage. Unfortunately, such a specialized test based on AC-equality alone is not sufficient.

Example 3.3 (ACI) *Consider the completion of* (A), (C), *and the idempotence axiom* (I) $x+x=x$. *Let* $\succ$ *be some LPO or some KBO. Using only the mentioned redundancy test, we get additionally to* (ACC') *the rule* $x+x \to x$ *and for each* $n \geq 2$*:*

$$\begin{array}{l} x_1 + \ldots + x_{n-1} + x_n + x_1 \to x_1 + \ldots + x_{n-1} + x_n \\ x_1 + \ldots + x_{n-1} + x_1 + x_n \to x_1 + \ldots + x_{n-1} + x_n \end{array}$$

Stronger tests are able to show all these rules (except one) to be ground joinable for $n \geq 2$. *So the completion stops with the finite ground convergent system*

$$\begin{array}{rclcrcl} (x+y)+z & \to & x+(y+z) & & & & \\ x+y & = & y+x & & x+(y+z) & = & y+(x+z) \\ x+x & \to & x & & x+(x+y) & \to & x+y \end{array}$$

In Chapter 6 we will describe generic tests based on different forms of constraints.

Up to now we have focused on the completion to obtain ground convergent systems. As we have seen the completion process stops more often successfully with a ground convergent system when we enrich UKB by additional redundancy criteria. Having available a ground convergent system for some theory allows not only to prove theorems easily but also to reject invalid conjectures. In some sense the original unfailing completion procedure is too conservative in redundancy elimination aiming more at confluence than at ground confluence.

For theorem proving we often encounter specifications with an undecidable word problem. Hence the completion cannot stop. Nevertheless, detecting more equations redundant pays off. In this situation it is sometimes beneficial to keep redundant equations for simplification, because having a stronger simplification relation may avoid subsequent work. Consider a redundant equation that occurs several times as normal form of different critical pairs. After the first (expensive) redundancy test the other instances can be deleted by simplification or unit-subsumption, which are usually much cheaper than the redundancy tests. Furthermore, the simplification relation extends this to arbitrary instantiations and contexts, so this applies to a much broader class of terms. Speaking in more abstract terms, an equation that has been shown redundant in an expensive way may itself be useful for cheap, rewrite-based redundancy proofs. As we will show in the next section the kept redundant equations need not be considered for computing critical pairs.

In the next example we consider a theorem proving problem that suggests to keep redundant equations.

Example 3.4 (Abelian Groups) *We consider the equations* (A), (C), $x+0=x$, *and* $x+-x=0$, *and we want to prove that* $x+my+-x=my$ *is a logical consequence.*

For that we use the Skolem constants a and b and use as reduction ordering an LPO based on the precedence $- >_{\mathcal{F}} + >_{\mathcal{F}} 0 >_{\mathcal{F}} a >_{\mathcal{F}} b$. *Then, no finite ground confluent rewrite system* $(R, E, \succ)$ *exists: Because of* $-a \succ -b \succ a \succ b$, *the occurrences of* $-a$ *and* a *within sums cannot be sorted to appear immediately together, but may appear at arbitrary distances. So we have to saturate the given equations to* $(R, E, \succ)$ *such that either*

$$(i) \quad a + mb + -a \downarrow mb \qquad or \qquad (ii) \quad b + ma + -b \downarrow ma$$

holds. (Note that in general it is totally unclear how to order the Skolem constants a and b!) Completing the equations while omitting ground joinable equations produces (ACC') *and*

$$\begin{array}{rcl@{\qquad}rcl} x + 0 & \to & x & 0 + x & \to & x \\ -0 & \to & 0 & x + -x & \to & 0 \\ --x & \to & x & -(x+y) & \to & -y + -x \end{array}$$

and for each $n \geq 2$

$$\begin{array}{rcl} x_1 + \ldots + x_{n-1} + x_n + -x_1 & \to & x_2 + \ldots + x_n \\ x_1 + \ldots + x_{n-1} + -x_1 + x_n & \to & x_2 + \ldots + x_n \end{array}$$

There is a big difference in the effort to prove (i) or (ii). For (i) we need only the rule $x_1 + \ldots + x_n + -x_1 \to x_2 + \ldots + x_n$ *for* $n = 3$, *but for (ii) we need it for* $n = m + 1$. *Consider the case* $m = 7$. *Our implementation of standard UKB needs less than half a second for (i), but more than half an hour for (ii). For the case* $m = 8$ *it still needs less than half a second for (i), but cannot prove (ii) within two days! It needs so much time because it computes a huge set of (redundant) equations before it finds the crucial rule. Our implementation using ground joinability criteria avoids this and proves (ii) in less than one second in both cases if it keeps redundant equations for simplification.*

Keeping redundant equations for simplification is beneficial in this example. If our implementation deletes all ground joinable equations, for the case $m = 7$ *it needs about 280 seconds to prove (ii), and for the case* $m = 8$ *about 400 seconds. It then spends most of the time in the ground joinability tests, often repeated for equations occurring many times.*

So, keeping redundant equations for simplification can help to stabilize a prover against undesired influences of the reduction ordering. Nevertheless, the influence of the reduction ordering must not be underestimated:

EXAMPLE 3.5 (ABELIAN GROUPS, REVISITED) *Changing the precedence of the LPO to* $+ >_{\mathcal{F}} - >_{\mathcal{F}} 0$, *standard unfailing completion diverges as well. In contrast, our implementation employing ground joinability as redundancy criterion finds a finite ground confluent system consisting of* (ACC') *and*

$$\begin{array}{rcl@{\qquad}rcl} x + 0 & \to & x & 0 + x & \to & x \\ -0 & \to & 0 & --x & \to & x \\ x + -x & \to & 0 & & & \\ x + -(x+y) & \to & -y & -x + y & \to & -(x + -y) \\ -(x + -y) & = & y + -x & x + -(y+z) & = & -(y + -(x + -z)) \end{array}$$

This implies a decision procedure for Abelian Groups, which can easily show logical consequences such as $x + my + -x = my$ for even greater values of m. For $m = 7$ it needs less than one millisecond for variants (i) and (ii), for $m = 1000$ less than one second for both variants. Note that there is no need for AC-matching such as in [PS81] or special requirements for the Skolem constants in the precedence $>_{\mathcal{F}}$ as in [MN90].

In the next section we develop the theoretical framework of our work. It establishes that we are allowed not only to delete redundant equations, but also to keep them for simplification without the need to consider them for computing critical pairs. Concrete redundancy criteria are the topic of Chapter 6 where we will give their definition, comment on their implementation, and evaluate their practical performance.

3.2 The inference system $\mathcal{G}$

In this chapter we show that our approach is sound and complete: We present an inference system and show by using the technique of proof transformations that the answer "yes" can be produced iff $u = v$ is a logical consequence of E. Our inference system is based on that of [BDP89] and refines the work presented in [AHL03].

The inference system works on triples (R, E, G) and uses a ground reduction ordering $\succ$. The intention is that $R \subseteq \succ$ stores the rules, E contains the unoriented equations, and G contains equations whose ground instances have a smaller proof in $(R, E, \succ)$. Equations in G will be used for the simplification of R and E, but not for computing critical pairs. We do not simplify equations in G.

The completeness of the approach is shown with the technique of proof transformations, with a *proof ordering* $\succ_{\mathcal{P}}$ as main ingredient [BDH86]. For all proofs P that are modified by some operation on (R, E, G) we have to ensure that there is some proof P' with $P \succcurlyeq_{\mathcal{P}} P'$. Let $s, t \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$. A (justified) *ground proof* for $s = t$ in (R, E, G) is of the form

$$P = (t_0, \varrho_1, t_1, \ldots, \varrho_n, t_n)$$

where $t_i \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$, $t_0 \equiv s$, and $t_n \equiv t$. For each *proof step* $t_{i-1}\ \varrho_i\ t_i$ the *justification* ϱ_i records the direction of the rewrite step, its position p, and the used equation $u = v$ and substitution σ. We define the complexity $c(s\ \varrho\ t)$ of a proof step by

$$c(s\ \varrho\ t) = \begin{cases} (\{s\}, s|_p, (l,r), t) & \text{if } s \longrightarrow_R t & \text{with } l \to r \text{ at } p \in \mathcal{O}(s) \\ (\{t\}, t|_p, (l,r), s) & \text{if } s \longleftarrow_R t & \text{with } l \to r \text{ at } p \in \mathcal{O}(t) \\ (\{s\}, s|_p, (u,v), t) & \text{if } s \longrightarrow_{(E\cup G)^{\succ}} t & \text{with } \sigma(u) \to \sigma(v) \text{ at } p \in \mathcal{O}(s) \\ (\{t\}, t|_p, (u,v), s) & \text{if } s \longleftarrow_{(E\cup G)^{\succ}} t & \text{with } \sigma(u) \to \sigma(v) \text{ at } p \in \mathcal{O}(t) \\ (\{s,t\}, -, -, -) & \text{if } s \longmapsto\!\dashv_{E\cup G} t & \text{is an unoriented step} \end{cases}$$

On these tuples we define ordering $\succ_t$ as the lexicographic combination $(\succ\!\!\succ, \succ_{\mathrm{st}}, \rhd, \succ)$. Here $\succ\!\!\succ$ is the multiset extension of the fixed ground reduction ordering $\succ$, $\succ_{\mathrm{st}}$ is the subterm ordering, and $\rhd$ is a well-founded ordering on term pairs that we will explain in the following paragraph. Let $\succ\!\!\succ_t$ be the multiset extension of $\succ_t$. We define the complexity $c(P)$ of ground proof P to be the multiset

$$c(P) = \{c(t_{i-1}\ \varrho_i\ t_i) \mid i = 1, \ldots, n\}$$

and an ordering $\succ_{\mathcal{P}}$ on ground proofs by

$$P_1 \succ_{\mathcal{P}} P_2 \quad \text{iff} \quad c(P_1) \succ\!\succ_t c(P_2) \ .$$

By construction $\succ_{\mathcal{P}}$ is a *proof ordering.* Especially it is well-founded and monotonic with respect to the proof structure, that is: $P \succ_{\mathcal{P}} P'$ implies $P_1 P P_2 \succ_{\mathcal{P}} P_1 P' P_2$.

To define the ordering $\rhd$, we use the following function Ψ to map term pairs to tuples. We have

$$\Psi(s,t) = (|s|, M(s), s, n),$$

where $M(s)$ gives the multiset of function symbols in s, and n indicates whether the term pair refers to a rule $s \rightarrow t$ in R, then $n = 1$, or to an equation $s \doteq t$ in $E \cup G$, then $n \in \{2,3,4\}$. The three values allow to differentiate between three kinds of equations. The first distinction is between *active* and *passive* equations, a division that is made by our main loop design (see Chapter 7): Active equations are used for simplification and for computing critical pairs, passive equations are critical pairs that are subject to simplification and are not yet activated. We use $n = 3$ for active equations and $n = 4$ for passive equations. For a fixed number of "distinguished" equations, such as C or C', we use $n = 2$. On these tuples we define the ordering $\rhd_{\Psi}$ as the lexicographic combination

$$\rhd_{\Psi} = (>_{\mathbb{N}}, \supset, \blacktriangleright, >_{\mathbb{N}}),$$

where $\blacktriangleright$ denotes the encompassment ordering. We then define $\rhd$ by

$$(s,t) \rhd (u,v) \quad \text{iff} \quad \Psi(s,t) \rhd_{\Psi} \Psi(u,v).$$

By $\triangleq$ we denote the corresponding equivalence: $(s,t) \triangleq (u,v)$ iff $(s,t) \unrhd (u,v)$ and $(u,v) \unrhd (s,t)$. This means that $(s,t) \triangleq (u,v)$ if s is a variable renaming of u and if both term pairs refer to rules or to equations of the same kind.

Note that by construction $\rhd$ is a well-founded ordering. It extends and refines the encompassment ordering $\blacktriangleright$ which is usually used to compare the left-hand sides of rules and equations. If $s \blacktriangleright l$ then $(s,t) \rhd (l,r)$. The value n captures the use of the symbol *max* in the proof ordering of [BDP89]: rewrite steps with rules are smaller than rewrite steps with equations. Therefore, the use of $\rhd$ makes explicit the ordering on rules and equations that is only implicit in the proof ordering of [BDP89]. Furthermore, ordering $\rhd$ better reflects the invariants maintained by an implementation and therefore justifies typical optimizations, such as the omission of explicit ordering tests[3]. However, the main advantage of this admittedly complex and technical proof ordering is that it extends the proof ordering of [BDP89] in a conservative way. Hence more proofs are comparable, which leads to more redundant equations and thus to stronger redundancy criteria. To understand some of the fine points of ordering $\rhd$ some background knowledge of the redundancy criteria is necessary. We therefore refer to Chapter 6, especially the explanations on pp. 130–131.

The following notion of redundancy is based on the proof ordering: an equation is redundant if every ground instance has a smaller proof.

[3] For example, we are not aware of any implementation that explicitly performs the $\blacktriangleright$-tests in the simplification of newly generated critical pairs.

DEFINITION 3.1 *We write* $s = t \succ_{\mathcal{P}} R(E \cup G)$ *if for any* $\sigma \in \mathrm{GSub}(s,t)$ *either* $\sigma(s) \equiv \sigma(t)$ *or there is a ground proof* P *for* $\sigma(s) = \sigma(t)$ *in* (R, E, G) *with* $\sigma(s) \longrightarrow_{\{s=t\}^{\succ}} \sigma(t) \succ_{\mathcal{P}} P$ *if* $\sigma(s) \succ \sigma(t)$, *or* $\sigma(s) \longleftarrow_{\{s=t\}^{\succ}} \sigma(t) \succ_{\mathcal{P}} P$ *if* $\sigma(t) \succ \sigma(s)$.

In case of this redundancy, the equation at hand is not needed for these smaller proofs:

LEMMA 3.1 *Let* $s = t \succ_{\mathcal{P}} R(E \cup G \cup \{s = t\})$. *Then for each ground proof* $\sigma(s)\ \varrho\ \sigma(t)$ *there is a proof* P_0 *in* (R, E, G) *not using the equation* $s = t$ *such that* $\sigma(s)\ \varrho\ \sigma(t) \succ_{\mathcal{P}} P_0$. *Especially,* $s = t \succ_{\mathcal{P}} R(E \cup G)$.

PROOF By assumption there is for any $\sigma \in \mathrm{GSub}(s,t)$ a proof P for $\sigma(s) = \sigma(t)$ such that $\sigma(s)\ \varrho\ \sigma(t) \succ_{\mathcal{P}} P$. Proof P may contain steps $\sigma'(s)\ \varrho'\ \sigma'(t)$ using $s = t$ itself. We show by induction on $\succ_{\mathcal{P}}$ that there is also a proof in (R, E, G), i. e., without such steps. Assume there are steps $\sigma'(s)\ \varrho'\ \sigma'(t)$ in P. Because of $\sigma(s)\ \varrho\ \sigma(t) \succ_{\mathcal{P}} P \succeq_{\mathcal{P}} \sigma'(s)\ \varrho'\ \sigma'(t)$, the induction hypothesis applies so that the steps $\sigma'(s)\ \varrho'\ \sigma'(t)$ can be replaced by smaller proofs in (R, E, G). So there is a proof P_0 in (R, E, G) such that $\sigma(s)\ \varrho\ \sigma(t) \succ_{\mathcal{P}} P \succ_{\mathcal{P}} P_0$. □

We now present the inference system $\mathcal{G}$.

DEFINITION 3.2 *The inference system* $\mathcal{G}$ *consists of the following inferences.*

1. DEDUCE
$$\frac{(R, E, G)}{(R, E \cup \{s = t\}, G)}$$
if $s =_{R \cup E \cup G} t$

2. ORIENT
$$\frac{(R, E \cup \{s \doteq t\}, G)}{(R \cup \{s \rightarrow t\}, E, G)}$$
if $s \succ t$

3. R-SIMPLIFY-RULE
$$\frac{(R \cup \{s \rightarrow t\}, E, G)}{(R \cup \{s \rightarrow u\}, E, G)}$$
if $t \longrightarrow_{R(E \cup G)} u$

4. L-SIMPLIFY-RULE
$$\frac{(R \cup \{s \rightarrow t\}, E, G)}{(R, E \cup \{u = t\}, G)}$$
if $s \longrightarrow_{R(E \cup G)} u$ *with* $l = r$ *and* $(s,t) \rhd (l,r)$

5. SIMPLIFY-EQUATION
$$\frac{(R, E \cup \{s \doteq t\}, G)}{(R, E \cup \{u = t\}, G)}$$
if $s \longrightarrow_{R(E \cup G)} u$ *with* $l = r$ *and* $t \succ s$ *or* $(s,t) \rhd (l,r)$

6. Subsume

$$\frac{(R, E \cup \{s = t\}, G)}{(R, E, G)}$$

if $s \longmapsto_{E \cup G} t$ *with* $l = r$ *and* $(s,t) \rhd (l,r)$

7. Detect

$$\frac{(R, E \cup \{s = t\}, G)}{(R, E, G \cup \{s = t\})}$$

if $s = t \succ_{\mathcal{P}} R(E \cup G)$

8. Delete

$$\frac{(R, E, G \cup \{s = t\})}{(R, E, G)}$$

We write $(R, E, G) \vdash (R', E', G')$ if (R', E', G') can be deduced by $\mathcal{G}$ from (R, E, G) in one step. A *$\mathcal{G}$-derivation* is a sequence $(R_i, E_i, G_i)_{i \in \mathbb{N}}$ such that $(R_i, E_i, G_i) \vdash (R_{i+1}, E_{i+1}, G_{i+1})$ for all $i \geq 0$. It is a *$\mathcal{G}$-derivation for E* if $(R_0, E_0, G_0) = (\varnothing, E, \varnothing)$.

Let us comment on several inferences. Inference Deduce expresses that we are free to add any equational consequence of (R, E, G) to E. For fairness reasons it is only necessary to do that for $\langle s, t \rangle \in \mathrm{CP}(R, E)$. Inference Delete expresses that it is not required to keep redundant equations. If we restrict the test in inference Detect to syntactic equality, i. e., $s \equiv t$, we get unfailing completion as in [BDP89] which restricts Delete to the case $s = s$. For inferences L-Simplify-rule, Simplify-equation, and Subsume explicit $\rhd$-tests can be avoided in practice. The definition of $\rhd$ takes into account the structure and invariants of our main completion loop. Therefore, the existence of a match implies the $\rhd$-relation on equations.

Our next lemma states that $\mathcal{G}$ is sound.

Lemma 3.2 *Let $(R_i, E_i, G_i)_{i \in \mathbb{N}}$ be a $\mathcal{G}$-derivation for E. Then for all i we have $R_i \subseteq \succ$ and $=_E\, = \,=_{R_i \cup E_i \cup G_i}$ on* $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$.

Proof Induction on i. For $i = 0$ both properties hold trivially. Only two inferences affect R_{i+1}. In case of Orient the side condition entails $R_{i+1} \subseteq \succ$. In case of R-Simplify-rule recall that $\longrightarrow_{R(E \cup G)} \subseteq \succ$ and that $\succ$ is transitive.

For the second property note that all new equations in $(R_{i+1}, E_{i+1}, G_{i+1})$ are equational consequences of (R_i, E_i, G_i). Furthermore, all proof steps using equations $s = t$ deleted by simplification or subsumption inferences can be replaced by one or two steps in $(R_{i+1}, E_{i+1}, G_{i+1})$. Inference Delete remains. When equation $s = t$ was inserted into G_j at time $j \leq i$ each ground step $\sigma(s)\ \varrho\ \sigma(t)$ had a smaller proof. By induction hypothesis there is still a proof for $\sigma(s)\ \varrho\ \sigma(t)$ in (R_i, E_i, G_i) and by Lemma 3.1 there is one without using $s = t$. So deleting $s = t$ preserves the equality relation on ground terms. □

Note that when inference Delete is restricted to the case $s = s$ (such as is done in [BDP89]) the equality relation is not only preserved on ground terms but on all terms.

The next lemma states that $\mathcal{G}$ is proof decreasing.

LEMMA 3.3 *Let $(R_i, E_i, G_i)_{i\in\mathbb{N}}$ be a $\mathcal{G}$-derivation. For any ground proof P for $v=w$ in (R_i, E_i, G_i) there is a ground proof P' for $v=w$ in $(R_{i+1}, E_{i+1}, G_{i+1})$ with $P \succcurlyeq_{\mathcal{P}} P'$.*

PROOF Induction on i. Because $\succ_{\mathcal{P}}$ is monotonic with respect to the proof structure it suffices to consider only proofs that consist of single proof steps affected by the application of a single inference. As $\succ$ is total on ground terms, unoriented proof steps of the form $v \longmapsto_{E\cup G} w$ do only occur for $v \equiv w$. We can replace such proof steps by the empty proof which is smaller. For oriented proof steps we proceed by case analysis (omitting symmetric cases):

1. DEDUCE. Because P is still a proof in $(R_{i+1}, E_{i+1}, G_{i+1})$ we can choose $P' = P$. But note that in case P is a peak $v \longleftarrow_{R(E\cup G)} u \longrightarrow_{R(E\cup G)} w$ that is covered by the deduced equation $s=t$, we can give a proof $P' = (v\ \varrho\ w)$ using $s=t$ such that $P \succ_{\mathcal{P}} P'$.
2. ORIENT. Let $s=t$ be oriented into rule $s\rightarrow t$ and P be an arbitrary proof step $v \longrightarrow_{\{s=t\}^{\succ}} w$ using the equation. By the orientation step, P has to be replaced by P' which is $v \longrightarrow_{\{s\rightarrow t\}} w$. We have to show that $P \succ_{\mathcal{P}} P'$ holds. The complexity is $(v, v|_p, (s,t), w)$ in both cases, however, the third component actually decreases with respect to $\rhd$: The last component of $\Psi(s,t)$ becomes smaller because (s,t) refers in the first proof to an equation and in the second proof to a rule.
3. R-SIMPLIFY-RULE. Let $t \longrightarrow_{R(E\cup G)} u$ with $l=r$ at p and P be an arbitrary proof of the form $v \longrightarrow_{\{s\rightarrow t\}} w$. We replace P be proof P' which is given as $v \longrightarrow_{\{s\rightarrow u\}} w' \longleftarrow_{\{l=r\}} w$. Then $P \succ_{\mathcal{P}} P'$, because $s \succ t \succ u$, hence $v \succ w \succ w'$ and thus $(v, v|_q, (s,t), w) \succ_t (w, w|_{q.p}, (l,r), w')$ by the first component. Furthermore, $(s,t) \triangleq (s,u)$ which implies $(v, v|_q, (s,t), w) \succ_t (v, v|_q, (s,u), w')$ by the last component.
4. L-SIMPLIFY-RULE. Let $s \longrightarrow_{R(E\cup G)} u$ with $l=r$ at p and P be some arbitrary proof $v \longrightarrow_{\{s\rightarrow t\}} w$. For its replacement P' we have to consider three different proofs depending on the ordering relation: Either $v \longrightarrow_{\{l=r\}} v' \longleftarrow_{\{u=t\}^{\succ}} w$, or $v \longrightarrow_{\{l=r\}} v' \longrightarrow_{\{u=t\}^{\succ}} w$, or $v \longrightarrow_{\{l=r\}} v' \longmapsto_{\{u=t\}} w$. The (identical) left steps are strictly smaller than $v \longrightarrow_{\{s\rightarrow t\}} w$, because $p \neq \lambda$ or $(s,t) \rhd (l,r)$ and therefore $(v, v|_q, (s,t), w) \succ_t (v, v|_{q.p}, (l,r), v')$ by the second or third component. The three different right steps are smaller by the first component of their complexity because $v \succ w$ and $v \succ v'$. So we have in each of the three cases a smaller proof.
5. SIMPLIFY-EQUATION. The case $t \succ s$ is analogous to the case R-SIMPLIFY-RULE; the case $(s,t) \rhd (l,r)$ is analogous to the case L-SIMPLIFY-RULE.
6. SUBSUME. Let $s \longmapsto_{E\cup G} t$ with $l=r$ at p and P be some arbitrary proof $v \longrightarrow_{\{s=t\}^{\succ}} w$. We replace P be proof P' which is $v \longrightarrow_{\{l=r\}^{\succ}} w$. Then $P \succ_{\mathcal{P}} P'$, because $(s,t) \rhd (l,r)$ and therefore $(v, v|_q, (s,t), w) \succ_t (v, v|_{q.p}, (l,r), w)$ by the second or third component.
7. DETECT. Ground proof P is still a proof in $(R_{i+1}, E_{i+1}, G_{i+1})$. However, by $s=t \succ_{\mathcal{P}} R(E\cup G)$ we know that there exists an even smaller ground proof P'.

8. Delete. Let $v \longrightarrow_{\{s=t\}^{\succ}} w$ with instance $\sigma(s) \to \sigma(t)$. When $s = t$ was transferred from E_j to G_{j+1}, $j < i$, there was for $\sigma(s) = \sigma(t)$ a proof P_0 that was smaller than $\sigma(s) \longrightarrow_{\{s=t\}^{\succ}} \sigma(t)$ By induction hypothesis and Lemma 3.1, there exists for $\sigma(s) = \sigma(t)$ a proof P_0' in (R_i, E_i, G_i) without $s = t$ such that $P_0 \succsim_{\mathcal{P}} P_0'$. Thus, there is a proof P' in $(R_{i+1}, E_{i+1}, G_{i+1})$ for $v = w$ that is smaller than $v \longrightarrow_{\{s=t\}^{\succ}} w$.

□

The next lemma is crucial for our approach. It states that any G_i-step in a ground proof can be replaced by a smaller proof in $(R_i, E_i, \varnothing)$. This can be considered as an extension of Lemma 3.1 to all elements of G. Note that during a $\mathcal{G}$-derivation cyclic dependencies may occur. For example, an equation $s = t$ in G simplifies a rule $l \to r$ that was used to shift $s = t$ from E to G.

Lemma 3.4 *Let $(R_i, E_i, G_i)_{i \in \mathbb{N}}$ be a $\mathcal{G}$-derivation for E. For all $i \in \mathbb{N}$ we have: For each ground proof P in (R_i, E_i, G_i) there is a ground proof P' in $(R_i, E_i, \varnothing)$ with $P \succsim_{\mathcal{P}} P'$.*

Proof Induction on $\succ_{\mathcal{P}}$. If P does not contain G_i-steps then let $P' = P$, as P is already a proof in $(R_i, E_i, \varnothing)$. Otherwise, let $\sigma(s)\ \varrho\ \sigma(t)$ be a proof step in P with $s = t$ in G_i. When $s = t$ was transferred from E_j to G_{j+1}, $j < i$, there was for $\sigma(s)\ \varrho\ \sigma(t)$ a ground proof P_0 that was strictly smaller than $\sigma(s)\ \varrho\ \sigma(t)$. By Lemma 3.3 there is a ground proof P_1 in (R_i, E_i, G_i) for $\sigma(s)\ \varrho\ \sigma(t)$ with $P_0 \succsim_{\mathcal{P}} P_1$. As $P \succsim_{\mathcal{P}} \sigma(s)\ \varrho\ \sigma(t) \succ_{\mathcal{P}} P_0 \succsim_{\mathcal{P}} P_1$ there is by induction hypothesis a proof P_1' in $(R_i, E_i, \varnothing)$ with $P_1 \succsim_{\mathcal{P}} P_1'$. Let P' result from P by replacing all G_i-steps by their corresponding proofs in $(R_i, E_i, \varnothing)$. Then P' is a proof in $(R_i, E_i, \varnothing)$ with $P \succ_{\mathcal{P}} P'$, as at least one G_i-step is replaced. □

A straightforward corollary is that in any $\mathcal{G}$-derivation G_i is not needed to preserve equality on ground terms.

Corollary 3.1 *Let $(R_i, E_i, G_i)_{i \in \mathbb{N}}$ be a $\mathcal{G}$-derivation for E. If $u, v \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ and $u =_E v$ then there exist ground proofs P_i of $u = v$ in $(R_i, E_i, \varnothing)$ with $P_i \succsim_{\mathcal{P}} P_{i+1}$ for all $i \in \mathbb{N}$. So we have $=_E \; = \; =_{R_i \cup E_i}$ on $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ for all $i \in \mathbb{N}$.* □

This motivates the definition of a fair $\mathcal{G}$-derivation: Equations in G_i need not be considered for computing critical pairs because for each ground proof with G_i-steps, there is a smaller one without.

Definition 3.3 *Let $(R_i, E_i, G_i)_{i \in \mathbb{N}}$ be a $\mathcal{G}$-derivation and let*

$$R^\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} R_j \qquad E^\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} E_j \qquad G^\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} G_j$$

be the set of persistent rules, persistent equations, and persistent redundant equations, respectively. The $\mathcal{G}$-derivation is fair *if $\mathrm{CP}(R^\infty, E^\infty) \subseteq \bigcup_{i \geq 0} E_i$.*

This definition adopts the usual notion of fairness to inference system $\mathcal{G}$. It is possible to use weaker notions, such as the following: A $\mathcal{G}$-derivation is fair if for any ground peak $u \longleftarrow v \longrightarrow v$ in $R^\infty(E^\infty)$ there is some $i \in \mathbb{N}$ such that there is a strictly smaller proof in (R_i, E_i, G_i).

Now we can formulate the main theorem of this chapter.

THEOREM 3.1 *Let $(R_i, E_i, G_i)_{i\in\mathbb{N}}$ be a fair $\mathcal{G}$-derivation for E.*

(a) $R^\infty(E^\infty)$ is ground confluent and terminating.
(b) For any $u, v \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$, if $u =_E v$ then $u \downarrow v$ in $R_i(E_i)$ for some i.

PROOF Let $u, v \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ and $u =_E v$. By Corollary 3.1 there is a sequence $P_0, P_1, P_2, \ldots$ of proofs for $u = v$ in $(R_i, E_i, \varnothing)$ with $P_i \succcurlyeq_{\mathcal{P}} P_{i+1}$. As $\succ_{\mathcal{P}}$ is well-founded this sequence has a minimum, say P_k. Because of the fairness condition P_k can have no peak, so it is of the form $u \xrightarrow{*} \cdot \xleftarrow{*} v$. As $P_k = P_{k+j}$ for all $j \geq 0$ this is also a proof in $R^\infty(E^\infty)$. Therefore $R^\infty(E^\infty)$ is ground confluent and we have $=_E\ =\ =_{R^\infty \cup E^\infty}$ on ground terms. The relation $R^\infty(E^\infty)$ is terminating because of $R^\infty(E^\infty) \subseteq \succ$. □

This theorem rises the following question: Why should redundant equations be kept at all as they are not needed to prove $u \downarrow v$ in some $R_i(E_i)$? The answer is that $R^\infty(E^\infty \cup G^\infty)$ is ground convergent as well and that when $u \downarrow v$ in $R_j(E_j \cup G_j)$ at point j, in practice this is typically with j smaller than i. Keeping redundant equations, although theoretically not necessary, allows a stronger and cheaper simplification relation and is therefore beneficial for the practical performance of a prover.

* * *

For an implementation of inference system $\mathcal{G}$ we need an algorithm for inference DETECT: The main problem is how to check whether $s = t \succ_{\mathcal{P}} R(E \cup G)$ holds. This property is undecidable in general – even if s and t are ground terms.

THEOREM 3.2 *For given rewrite system $(R, E, \succ)$ and ground equation $s = t$ it is undecidable whether $s = t \succ_{\mathcal{P}} R(E)$ holds.*

PROOF It is well known that the ground word problem is undecidable in general (see [BN98, p. 60] for examples). That is in our terms, whether for given E and equation $s = t$ with $s, t \in \mathrm{Term}(\mathcal{F})$ there is some proof P in E for $s = t$. We will show that this is equivalent to the test $f(s) = f(t) \succ_{\mathcal{P}} R(E)$. Here, f is a new unary function symbol and $R = \{f(x) \to x\}$. Note that $\succ_{\mathcal{P}}$ contains as parameter a reduction ordering $\succ$. For $\succ$ we choose an LPO with total precedence such that $f >_{\mathcal{F}} g$ for all $g \in \mathcal{F}$. Hence, $f(u) \succ v$ for all $u, v \in \mathrm{Term}(\mathcal{F})$. To prove the theorem we show:

There exists a ground proof P in E for $s = t$ if, and only if, $f(s) = f(t) \succ_{\mathcal{P}} R(E)$.

Let P be a ground proof in E for $s = t$ and (without loss of generality) $s \succ t$. Then $P_1 = f(s) \longrightarrow_R s\ P\ t \longleftarrow_R f(t)$ is a proof in $R(E)$ for $f(s) = f(t)$. Proof P_1 is strictly

smaller than proof $P_0 = f(s) \longrightarrow_{\{f(s)=f(t)\}^{\succ}} f(t)$ as all proof steps in P_1 are $\succ_t$-smaller than the single proof step in P_0: Because $f(s) \trianglerighteq f(x)$ and hence $(f(s), f(t)) \triangleright (f(x), x)$ the first proof step in P_1 is strictly smaller by the third component. The other proof steps are strictly smaller by the first component: $f(s) \succ f(t)$ and $f(s) \succ u$ for each term u in P as these ground terms do not contain f.

Let $f(s) = f(t) \succ_{\mathcal{P}} R(E)$. Then there is a proof P for $f(s) = f(t)$ in $(R, E, \succ)$ with $f(s) \longrightarrow_{\{f(s)=f(t)\}^{\succ}} f(t) \succ_{\mathcal{P}} P$. From P we construct a proof P' in the following way: Delete all steps using $f(x) \to x$ and erase in all terms all occurrences of f. Then P' is a proof for $s = t$ in E.

Hence, it is undecidable in general whether $s = t \succ_{\mathcal{P}} R(E)$ holds as otherwise the ground word problem were decidable which is not the case. □

In our construction it is essential for the choice of $\succ$ that $f(u) \succ v$ for all $u, v \in \mathrm{Term}(\mathcal{F})$, not that $\succ$ is some LPO. If, however, $\succ$ is a reduction ordering such that for any ground term u there is only a finite number of ground terms v with $u \succ v$ then the property $s = t \succ_{\mathcal{P}} R(E)$ becomes decidable for $s, t \in \mathrm{Term}(\mathcal{F})$. But the whole point of our approach is to show the redundancy of $s = t$ for nonground s and t. So we have to resort for inference DETECT to sufficient approximations of $s = t \succ_{\mathcal{P}} R(E \cup G)$. We will do this by a strengthened ground joinability test.

DEFINITION 3.4 *We define the restricted rewrite relation* $u \xrightarrow{s=t\,\triangleright}_{R(E\cup G)} v$ *as:*

$$u \xrightarrow{s=t\,\triangleright}_{R(E\cup G)} v \quad \textit{iff} \quad u \longrightarrow_{R(E\cup G)} v \textit{ with } l = r \textit{ in } (R, E, G) \textit{ at } p \textit{ and } p \neq \lambda \textit{ or } (s,t) \triangleright (l,r).$$

We define $s \Downarrow_{\triangleright} t$ *in* $R(E \cup G)$ *iff for any* $\sigma \in \mathrm{GSub}(s,t)$ *either*

(a) $\sigma(s) \equiv \sigma(t)$ *or*

(b) $\sigma(s) \succ \sigma(t)$ *and* $\sigma(s) \xrightarrow{s=t\,\triangleright}_{R(E\cup G)} s_1 \downarrow \sigma(t)$, *or*

(c) $\sigma(t) \succ \sigma(s)$ *and* $\sigma(t) \xrightarrow{t=s\,\triangleright}_{R(E\cup G)} t_1 \downarrow \sigma(s)$.

Note that in the previous definition the $\triangleright$-test is based on the uninstantiated term pairs (s,t) and (l,r), not on the concrete instances.

THEOREM 3.3 *If* $s \Downarrow_{\triangleright} t$ *in* $R(E \cup G)$ *then* $s = t \succ_{\mathcal{P}} R(E \cup G)$.

PROOF The case $\sigma(t) \equiv \sigma(s)$ is trivial. Assume $\sigma(s) \succ \sigma(t)$ and let P be the proof $\sigma(s) \xrightarrow{s=t\,\triangleright}_{R(E\cup G)} s_1 \xrightarrow{*} \cdot \xleftarrow{*} \sigma(t)$. All proof steps $u \ \varrho \ v$ in P are $\succ_t$-smaller than $\sigma(s) \longrightarrow_{\{s=t\}^{\succ}} \sigma(t)$: For the first proof step this is by the second or third component of its complexity. For all other proof steps this is by the first component because $\sigma(s)$ is the greatest term in P. The case $\sigma(t) \succ \sigma(s)$ is analogous. So we have $s = t \succ_{\mathcal{P}} R(E \cup G)$. □

It is indeed necessary to use the strengthened ground joinability test $s \Downarrow_{\triangleright} t$ instead of $s \Downarrow t$ for Theorem 3.3. Otherwise, we could use $s = t$ to show itself redundant and would immediately lose completeness. However, there are also examples without self-reference.

Example 3.6 *Let E be $\{a=b,\ a=c,\ a=d,\ a=e,\ b=c,\ d=e\}$. Let $\succ$ be the LPO for $a >_{\mathcal{F}} b >_{\mathcal{F}} c >_{\mathcal{F}} d >_{\mathcal{F}} e$. Using the test $s \Downarrow t$ in (R_i, E_i, G_i) we can successively apply* Detect *to equations $a=b$, $a=c$, $a=d$, and $a=e$. We have $R_4 = \varnothing$, $E_4 = \{b=c,\ d=e\}$, and $G_4 = \{a=b, a=c, a=d, a=e\}$. Therefore,* $\mathrm{CP}(R_4, E_4) = \varnothing$. *Although $b =_E e$, we have neither $b =_{R_4 \cup E_4} e$ nor $b \downarrow e$ in (R_4, E_4, G_4). Hence, replacing $\Downarrow_{\rhd}$ by $\Downarrow$ does not preserve the equality relation on ground terms; replacing $\Downarrow_{\rhd}$ by $\Downarrow$ makes Theorem 3.3 invalid.*

In contrast, for testing a given rewrite system for ground confluence the test $s \Downarrow t$ is sufficient [MN90, CNNR03]. This is similar to the test whether a rewrite system is confluent. There, too, the $\rhd$-test is not necessary. During completion, however, interreductions need a careful treatment. Otherwise completeness is endangered. In standard completion this is reflected by the $\rhd$-conditions for inferences L-Simplify-rule and Simplify-equation [BDP89].

Unfortunately, the tests $s \Downarrow t$ and $s \Downarrow_{\rhd} t$ are still undecidable. For ordered rewriting even the joinability of two terms is undecidable, which we will show next. This answers an open question of [CNNR03]. In the proof of the next theorem we make use of the following decision problem [HU79], which we will use in a construction similar to [KNO90, Chap. 6]:

Definition 3.5 *The* Modified Post Correspondence Problem *(MPCP) is the following. Given two finite lists $A = (u_1, \ldots, u_n)$ and $B = (v_1, \ldots, v_n)$ of words from Σ^+, does there exist a finite sequence $i_1, i_2, \ldots, i_k \in \{2, 3, \ldots, n-1\}$ such that*

$$u_1 u_{i_1} u_{i_2} \ldots u_{i_k} u_n = v_1 v_{i_1} v_{i_2} \ldots v_{i_k} v_n \ ?$$

The difference between MPCP and the usual PCP is that a solution is required to start with the first word on each list and end with the last word on each list. The MPCP is undecidable in general, because the halting problem of Turing machines can be reduced to this problem.

Theorem 3.4 *It is undecidable in general whether in a finite ordered rewrite system $(R, E, \succ)$ two ground terms s and t are joinable.*

Proof By reduction of MPCP. Let $A = (u_1, \ldots, u_n)$ and $B = (v_1, \ldots, v_n)$ be an arbitrary instance of MPCP over alphabet $\Sigma = \{a_1, \ldots, a_m\}$. Let $sig = (\mathcal{S}, \mathcal{F}, \alpha)$ with sorts $\mathcal{S} = \{S\}$ and operators $\mathcal{F} = \{f, g_1, \ldots, g_n, h, a_1, \ldots, a_m, b, c\}$, where f has arity four, b and c are constants, and all other function symbols are unary. The function symbols $a_1, \ldots, a_m$ are used to encode a word $u \in \Sigma^+$ by the term $\hat{u}(x)$ in the usual way. For example, if $u \equiv a_1 a_2 a_2$, then $\hat{u}(x) \equiv a_1(a_2(a_2(x)))$. The general idea is to represent a potential solution $i_1, i_2, \ldots, i_k \in \{2, 3, \ldots, n-1\}$ of (A, B) by the ground term $f(\hat{u}_1 \hat{u}_{i_1} \hat{u}_{i_2} \cdots \hat{u}_{i_k} \hat{u}_n(b),\ g_1 g_{i_1} g_{i_2} \cdots g_{i_k} g_n(b),\ \hat{v}_1 \hat{v}_{i_1} \hat{v}_{i_2} \cdots \hat{v}_{i_k} \hat{v}_n(b),\ b)$. The function symbols g_i, $i = 1, \ldots, n$, encode the integer sequence of the solution.

Let $\succ$ be the LPO to $h >_{\mathcal{F}} f >_{\mathcal{F}} g_1 >_{\mathcal{F}} \cdots >_{\mathcal{F}} g_n >_{\mathcal{F}} a_1 >_{\mathcal{F}} \cdots >_{\mathcal{F}} a_m >_{\mathcal{F}} b >_{\mathcal{F}} c$. Note that $h(s) \succ t$ for all $s \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$ and $t \in \mathrm{Term}(\mathcal{F} - \{h\})$. Especially, $h(c) \succ t$

iff $t \in \mathrm{Term}(\mathcal{F} - \{h\})$. The system $R \subseteq \succ$ consists of the following n rules:

$$\begin{aligned} f(\hat{u}_1(x), g_1(y), \hat{v}_1(z), b) &\to f(x, y, z, c) \\ f(\hat{u}_i(x), g_i(y), \hat{v}_i(z), c) &\to f(x, y, z, c) \qquad \text{for } i = 2, \ldots, n-1 \\ f(\hat{u}_n(b), g_n(b), \hat{v}_n(b), c) &\to c \end{aligned}$$

Because of the g_i, $i = 1, \ldots, n$, no critical pairs exist, hence R is confluent. It is easy to see that $\sigma(f(x, y, z, b)) \xrightarrow{*}_R c$ iff $\sigma(f(x, y, z, b))$ represents a solution to (A, B). The system E consists of equation $h(x') = f(x, y, z, b)$ only. Because of E, the ordered rewrite system $(R, E, \succ)$ is neither confluent nor ground confluent. For example, the term $h(b)$ rewrites with $E^{\succ}$ to the two irreducible terms $f(b, b, b, b)$ and $f(c, c, c, b)$.

Now consider the joinability in $(R, E, \succ)$ of the ground terms $s \equiv h(c)$ and $t \equiv c$. The term t is irreducible, for it is the smallest term with respect to $\succ$. The term s is R-irreducible, but ordered rewrite steps with $h(x') = f(x, y, z, b)$ are possible. As the equation contains extra variables, which are not determined by the match, we have $s \longrightarrow_{E^{\succ}} \sigma(f(x, y, z, b))$ for all substitutions σ such that $s \succ \sigma(f(x, y, z, b))$. This means that $\sigma(f(x, y, z, b)) \in \mathrm{Term}(\mathcal{F} - \{h\})$. Such terms are irreducible by $E^{\succ}$ and rewriting with R preserves this property. So the possible normal form derivations of s are of the form $s \longrightarrow_{E^{\succ}} \sigma(f(x, y, z, b)) \xrightarrow{!}_R s_\sigma$. As R is convergent the normal form s_σ is solely determined by σ. Especially, $s_\sigma \equiv c$ iff σ describes a solution to (A, B).

Hence, s and t are joinable in $(R, E, \succ)$ iff (A, B) has a solution. □

The construction of this theorem can be used to show several related decision problems to be undecidable.

Corollary 3.2 *Let $(R, E, \succ)$ be a finite ordered rewrite system and s, t, u be terms where u is in $R(E)$-normal form. Then it is undecidable in general whether $s \downarrow t$ (joinability), $s \Downarrow t$ (ground joinability), $s \xrightarrow{*}_{R(E)} t$ (reachability), or $s \xrightarrow{*}_{R(E)} u$ (normal form reachability) holds.* □

The theorem and its corollary might be surprising as ground confluence of an ordered rewrite system $(R, E, \succ)$ is decidable when $\succ$ is an LPO by the method described in [CNNR03]. The difference lies in the used ordered rewrite relation. For the confluence trees of [CNNR03] (see also Chapter 6.4) the relation $E_{\succ}$ suffices. For Theorem 3.4, however, we make extensive use of extra variables and their various instances contained in $E^{\succ}$. The undecidability results of [KNO90] are not directly applicable to our case, as they concern ground confluence (and ground joinability) with respect to $\mathrm{Term}(\mathcal{F})$, not with respect to $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$, and their constructions explicitly mention all function symbols.

Although the property $s \Downarrow t$ is undecidable, and therefore its strengthened form $s \Downarrow_{\rhd} t$ also is ($s \Downarrow_{\rhd} t$ iff $f(s) \Downarrow f(t)$ for some new operator f), sufficient redundancy criteria based on Theorem 3.3 are powerful in practice. This is the topic of Chapter 6.

3.3 Extensions and related work

The first extension we consider is the treatment of an arbitrarily quantified conjecture $(\forall\exists)^*.\, u = v$. In this case, Skolemization is more complicated than in the case of only universally quantified equations where each variable is simply replaced by a fresh Skolem constant. Recall that in refutational theorem proving the conjecture is added as the negated formula $\neg(\forall\exists)^*.\, u = v$. To get a Skolem normal form, the negation symbol is pushed inwards so that universal quantifiers are turned into existential ones and vice versa. Then the existentially quantified variables in $(\exists\forall)^*.\neg(u = v)$ are replaced by Skolem *functions* that have as arguments the variables that are universally quantified in an outer scope. For details we refer to the overview [NW01]. As result we get $\forall^*.\neg(\bar{u} = \bar{v})$ which is equivalent to $\neg\exists^*.\, \bar{u} = \bar{v}$.

To refute such an equation we need a solution for each variable in $\bar{u} = \bar{v}$ to make both sides of the equation E-equivalent. Hence we are looking for a substitution $\sigma \in \mathrm{GSub}(\bar{u}, \bar{v})$ such that $\sigma(\bar{u}) =_E \sigma(\bar{v})$. For convergent rewrite systems such substitutions can be enumerated by *narrowing* (see [Han94] for an overview). As E is not convergent initially, we have to interleave the narrowing process with the completion of E into R. This can be achieved by the following encoding [BDP89]: We introduce new constants *tt* and *ff* and a new binary operator *eq* and start the completion process with $E' = E \cup \{\, eq(x, x) = tt,\, eq(\bar{u}, \bar{v}) = ff\,\}$ and $tt = ff$ as conjecture. Then the standard unfailing completion process performs the search for substitution σ and the search for the ground convergent rewrite system R in parallel.[4]

In [Hil00] it is shown how to extend this technique to handle arbitrary formulas as conjecture. With a similar encoding, even axiomatizations described by Horn formulas (with equality) can be represented [BDP89]. We can therefore enhance the scope of a prover based on unfailing completion to a larger fragment of first-order logic with only moderate effort. For an extension to full clauses with equality, however, it seems more appropriate to develop a system based on the superposition calculus [BG94].

Completion and rewriting techniques for equational theorem proving go back to the seminal paper [KB70]. The authors note that, although they can improve on the work of [Eva51] in that they can cope with the associativity axiom, the handling of commutativity remains as an open problem. In [Plo72] it was suggested to build theories into the basic reasoning inferences. This was successfully pursued in the case of AC-theories in [LB77, PS81, JK86]. Especially the problem of unification modulo some theory has attained much interest, see [BS01] for a recent overview. From a practical point of view many of the results in this domain are not encouraging: Some theories have an undecidable unification problem, for others, the complexity is exponential or worse. To eliminate many of the doubly exponential number of AC-unifiers methods have been developed that do not affect completeness (e. g. [ZK90]), but in practice simpler and incomplete approaches are used [McC97b].

The ideas of unfailing completion can be traced back to [Lan75] and [Bro75]. The introduction of proof orderings [BDH86] helped to bring the approach into its final

[4] The experiences with the implementation of this method in WALDMEISTER show that equations with *eq* can be treated as normal equations throughout the whole system with one exception: Their heuristic assessment should be different to the one used for ordinary critical pairs.

form [BDP89, BD94]. The use of completion techniques as semi-decision procedures was mentioned in [Hue80] and later refined in [HR87]. This was further investigated in [BH95] where the authors concentrate on an adequate notion of fairness.

In [MN90] and [Pet90] constraint techniques are used to modify unfailing completion towards ground confluence. We refine the approach of [MN90] in Chapter 6.3 to get an efficient criterion for high-performance theorem provers. In [Bac91, p.83] Bachmair gives an inference system for ordered completion to achieve ground confluence only. This inference system could be used to establish correctness and completeness of our inference system $\mathcal{G}$. But this inference system is more of theoretical interest as it lacks any notion of effective redundancy criteria.[5] Furthermore, we use in $\mathcal{G}$ a refined $\rhd$-ordering on equations to strengthen the criteria, and the formulation with explicit G well reflects the use of redundant equations for simplification.

A different view on the role of G is to consider it as a form of critical pair criterion [BD88]. Critical pairs with elements of G are unnecessary as the peaks they cover have already smaller proofs. We prefer the more constructive view that elements of G are redundant, but enhance the simplification relation. In some sense, the use of G is even complementary to critical pair criteria. Although these criteria may avoid the generation of an equation, the very same equation may be generated by a different overlap and therefore enters the saturation loop despite its redundancy. Thus, critical pair criteria eliminate redundancy locally, whereas keeping redundant equations allows us to eliminate redundancy in a more global scope.

Much attendance has been given to the use of constraints for completion [KKR90, BGLS95, NR95], and recent overviews use constraint based formulations [GN01, NR01]. One might ask why we do not use an approach based on constraint inheritance. This is because then the simplification relation has to be weakened to ensure completeness. Natural formulations of constraint based simplification turned out to be undecidable [CT97]. In our experience this leads to far inferior systems in practice – at least for unit-equality problems, where simplification plays a dominant role [Ron03, Tür03]. The situation may be different for the full clausal case. In this case simplification is not as important and the restrictions of the generating inferences may pay off. One should note that for the frequently cited success with basicness constraints, the proof of the Robbins theorem [McC97b], simplification was performed in an incomplete manner[6].

The extension of rewriting and completion techniques to full clausal logic is done in the superposition calculus [BG94, BG98, NR01]. Here a semantic notion of redundancy is used: A clause C is redundant with respect to a set of clauses S if each ground instance $\sigma(C)$ is a logical consequence of smaller ground instances from elements of S. This notion is then used to justify practical redundancy elimination methods, such as simplification or subsumption. It is possible to extend our approach to the superposition calculus. However, one has to take into account the restrictions on top-level steps because the orderings used in the superposition calculus do not make such fine

[5] Even the seminal papers [Rob65] and [KB70] stress the importance of practical subsumption and simplification methods.

[6] For details see the file `basic.doc` in the EQP-distribution [McC05b]. McCune told us that the use of the super-0-strategy which discards most of the AC-unifiers was far more important than the basicness strategy. This is confirmed by our own experiments with EQP.

distinctions as our ordering $\succ_{\mathcal{P}}$ does. On the other side, full clauses give more freedom to use ground joinability as test for redundancy. Consider clause $C = \{s = t, u = v\}$. To show C redundant by ground joinability, it is not necessary to show $s \Downarrow t$ or $u \Downarrow v$, it is merely sufficient to show for each $\sigma \in \mathrm{GSub}(s, t, u, v)$ that $\sigma(s) \downarrow \sigma(t)$ or $\sigma(u) \downarrow \sigma(v)$ holds. This may compensate for the restrictions imposed by the ordering.

4 Orderings and their efficient implementation

For the implementation of a practically successful prover it is important to have efficient implementations of the most time-consuming subtasks. Whereas much research has been spent, for example, on the development of efficient indexing techniques (see [RSV01] for an overview), the implementation of orderings has received far less attention (a recent exception is [RV04]). However, the time that is spent on determining ordering relations between terms can amount to a significant part of the prover's overall running time. For example, WALDMEISTER [LH02] needs up to 50 % of the total running time for ordering comparisons, although a lot of effort went into the optimization of the corresponding routines and, with the default settings of WALDMEISTER, many ordering comparisons are avoided by using unorientable equations only in a very restricted form for rewriting [BH96].

The aim of this chapter is the development of efficient versions of the *Lexicographic Path Ordering* (LPO) [KL80] and the *Knuth-Bendix Ordering* (KBO) [KB70], two of the orderings in widespread use. Both are important for us because they are ground reduction orderings if the precedence is total. For LPO, several ideas lead from the direct translation of the usual definition, which has an exponential behavior, to a nonobvious implementation with polynomial requirements. In a similar way, the direct translation of the definition of KBO has quadratic behavior. From that we derive a version that runs in linear time.

Our approach is based on the methodology of *program transformations* [BD77, PP93]. In our case, this amounts to the following: We formulate the programs in a language that is close to functional programming or algebraic specification. This concise and abstract notation allows us to focus on the essential ideas in the development of the efficient version. We start with some "obviously correct" implementation, which is as close as possible to the original definition of the ordering. Then we refine the implementation in several small steps. The equivalence of two successive versions can be shown by a proof with induction. As the language used can easily be translated into the input language of some inductive theorem prover such as QUODLIBET [AKSSW03], these proofs can be performed within such a system. These formal proofs are more elaborate than manual proofs, but prevent the introduction of bugs by avoiding oversights.

To measure the progress between the different versions we translate them in a straightforward way into the programming language C and integrate them into WALDMEISTER. This allows us to test and compare them on thousands of test cases occurring in real proof-attempts and shows the impact of the different optimizations on a real prover running on real hardware.

The rest of this chapter is structured as follows: In Section 4.1 we discuss some preliminaries for the implementation. The development of efficient versions of LPO and KBO are the topics of Sections 4.2 and 4.3, respectively. Section 4.4 contains a brief discussion about related work and the implementation status in different systems. We conclude in Section 4.5.

4.1 Preliminaries for the implementation

We describe the different implementations in a small algebraic specification language. For boolean values it has a sort Bool with the two constructors $\mathsf{true}, \mathsf{false} :\rightarrow \mathsf{Bool}$. With their help we define disjunction and conjunction:

$$\begin{array}{rcl} \vee & : & \mathsf{Bool}\ \mathsf{Bool} \rightarrow \mathsf{Bool} \\ b_1 \vee b_2 & = & \textbf{if } b_1 \textbf{ then } \mathsf{true} \textbf{ else } b_2 \end{array} \qquad \begin{array}{rcl} \wedge & : & \mathsf{Bool}\ \mathsf{Bool} \rightarrow \mathsf{Bool} \\ b_1 \wedge b_2 & = & \textbf{if } b_1 \textbf{ then } b_2 \textbf{ else } \mathsf{false} \end{array}$$

This asymmetric definition expresses that the second argument is not evaluated if the result is already determined by the first. This is in conformance to many programming languages including C, Java, or Haskell. If all arguments are defined, both operations have the AC-property (i. e., the associativity and commutativity laws hold):

$$\begin{array}{rcl} (b_1 \vee b_2) \vee b_3 & = & b_1 \vee (b_2 \vee b_3) \\ (b_1 \wedge b_2) \wedge b_3 & = & b_1 \wedge (b_2 \wedge b_3) \end{array} \qquad \begin{array}{rcl} b_1 \vee b_2 & = & b_2 \vee b_1 \\ b_1 \wedge b_2 & = & b_2 \wedge b_1 \end{array}$$

We assume the availability of the sorts Vid and Fid to represent variables $\mathcal{V}$ and function symbols $\mathcal{F}$. Then we define the data type Term via two constructors using an additional sort $\mathsf{Termlist}$ for lists of terms.

$$\begin{array}{rcl} \mathsf{V} & : & \mathsf{Vid} \rightarrow \mathsf{Term} \\ \mathsf{F} & : & \mathsf{Fid}\ \mathsf{Termlist} \rightarrow \mathsf{Term} \end{array} \qquad \begin{array}{rcl} [\,] & : & \rightarrow \mathsf{Termlist} \\ \centerdot & : & \mathsf{Term}\ \mathsf{Termlist} \rightarrow \mathsf{Termlist} \end{array}$$

Note that for a term $\mathsf{F}(f, ts)$ there is no relationship encoded in the data type between the arity of f and the length of ts. It is therefore possible to construct Terms that are not well-formed, that is, they do not represent elements of $\mathrm{Term}(\mathcal{F}, \mathcal{V})$. An alternative is to have for each arity n a special constructor F_n, an approach which in our opinion is not feasible. To distinguish well-formed terms, we define the following predicates (or rather boolean valued functions). We assume the availability of the functions $\mathsf{arity} : \mathsf{Fid} \rightarrow \mathsf{Nat}$ and $\mathsf{length} : \mathsf{Termlist} \rightarrow \mathsf{Nat}$.

$$\begin{array}{rcl} \mathsf{well} & : & \mathsf{Term} \rightarrow \mathsf{Bool} \\ \mathsf{well}(\mathsf{V}(x)) & = & \mathsf{true} \\ \mathsf{well}(\mathsf{F}(f, ts)) & = & \mathsf{arity}(f) = \mathsf{length}(ts) \wedge \mathsf{well}_{\mathsf{tl}}(ts) \end{array} \qquad \begin{array}{rcl} \mathsf{well}_{\mathsf{tl}} & : & \mathsf{Termlist} \rightarrow \mathsf{Bool} \\ \mathsf{well}_{\mathsf{tl}}([\,]) & = & \mathsf{true} \\ \mathsf{well}_{\mathsf{tl}}(t \centerdot ts) & = & \mathsf{well}(t) \wedge \mathsf{well}_{\mathsf{tl}}(ts) \end{array}$$

The explicit use of the predicates is necessary in an inductive prover, which does not allow implicit assumptions, such as the requirement that all terms are well-formed. The definition of the auxiliary function $\mathsf{well}_{\mathsf{tl}}$ lifts well from Term to $\mathsf{Termlist}$. The alternative is the use of higher-order functions which would complicate the translation into the input language of some (first-order) inductive prover. This pattern of some function over Term and some auxiliary function over $\mathsf{Termlist}$, which call each other recursively, can also be observed in the next definition.

The function $\mathsf{contains}$ tests whether a variable symbol occurs in a term.

$$\begin{array}{rcl} \mathsf{contains} & : & \mathsf{Term\ Vid} \to \mathsf{Bool} \\ \mathsf{contains}(\mathsf{V}(x), y) & = & x = y \\ \mathsf{contains}(\mathsf{F}(f, ts), y) & = & \mathsf{contains_{tl}}(ts, y) \end{array} \qquad \begin{array}{rcl} \mathsf{contains_{tl}} & : & \mathsf{Termlist\ Vid} \to \mathsf{Bool} \\ \mathsf{contains_{tl}}([\,], y) & = & \mathsf{false} \\ \mathsf{contains_{tl}}(t \,\centerdot\, ts, y) & = & \mathsf{contains}(t, y) \\ & & \vee\ \mathsf{contains_{tl}}(ts, y) \end{array}$$

Obviously, the worst-case running time of $\mathsf{contains}$ and $\mathsf{contains_{tl}}$ is linear in the size of the first argument.

4.2 The Lexicographic Path Ordering

Let us first recall the definition of the LPO which requires $\mathcal{F}$ to contain function symbols of fixed arity only.

DEFINITION 4.1 *Let $>_\mathcal{F}$ be a partial ordering on $\mathcal{F}$ and $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. Then $s \succ_{\mathrm{lpo}} t$ iff either $s \equiv f(s_1, \ldots, s_n)$, $t \equiv g(t_1, \ldots, t_m)$, and*

(α) *$s_i \succeq_{\mathrm{lpo}} t$ for some $i \in \{1, \ldots, n\}$ or*
(β) *$f >_\mathcal{F} g$ and $s \succ_{\mathrm{lpo}} t_k$ for all $k \in \{1, \ldots, m\}$ or*
(γ) *$f = g$, there exists some $i \in \{1, \ldots, m\}$ such that $s_j \equiv t_j$ for all $j \in \{1, \ldots, i-1\}$ and $s_i \succ_{\mathrm{lpo}} t_i$, and $s \succ_{\mathrm{lpo}} t_k$ for all $k \in \{1, \ldots, m\}$*

or $s \equiv f(s_1, \ldots, s_n)$, $t \equiv x$, $x \in \mathcal{V}$ and

(δ) *$x \in \mathrm{Var}(s)$,*

where $u \succeq_{\mathrm{lpo}} v$ iff $u \equiv v$ or $u \succ_{\mathrm{lpo}} v$ for $u, v \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$.

Note that this definition follows e. g. [Ave95, BN98]: comparing terms to variables is captured in a separate case, namely case (δ). This does not only allow a more efficient implementation, it also better fits the pattern-matching approach used in our programming language. Some authors use the following version of this case:

(δ') $s_i \succeq_{\mathrm{lpo}} x$ for some $i \in \{1, \ldots, n\}$.

It is easy to see that both definitions are equivalent. With this different formulation it is possible to join cases (α) and (δ') into one combined case. Furthermore, for some extensions of LPO this variant is more appropriate (cf. e. g. Definition 5.4 on p. 84 or Lemma 6.5 on p. 120). However, the topic of this section is the development of an efficient implementation of LPO. Therefore, we use the formulation of Definition 4.1 in this chapter.

Before we continue, we want to restate some well-known properties of LPO:

PROPOSITION 4.1 *Let $>_\mathcal{F}$ be a precedence. Then the LPO $\succ_{\mathrm{lpo}}$ for $>_\mathcal{F}$ is a reduction ordering on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$. If $>_\mathcal{F}$ is total on $\mathcal{F}^{\mathrm{e}}$ then $\succ_{\mathrm{lpo}}$ is a ground reduction ordering. The subterm relation $\succ_{\mathrm{st}}$ is embedded in $\succ_{\mathrm{lpo}}$ (i. e., $\succ_{\mathrm{st}} \subseteq \succ_{\mathrm{lpo}}$).* □

The following program tries to follow Definition 4.1 as closely as possible. We call it the *reference implementation.*[1] Functions $>_{\mathsf{F}}$, $=_{\mathsf{t}}$, and $=_{\mathsf{tl}}$ compute the precedence, syntactic equality on Term, and syntactic equality on Termlist.

$$
\begin{array}{rcl}
\mathsf{lpo}_1 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Bool} \\
\mathsf{lpo}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \mathsf{alpha}_1(ss, \mathsf{F}(g, ts)) \vee \mathsf{beta}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \\
 & & \vee\ \mathsf{gamma}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \\
\mathsf{lpo}_1(\mathsf{F}(f, ss), \mathsf{V}(y)) & = & \mathsf{delta}_1(\mathsf{F}(f, ss), \mathsf{V}(y)) \\
\mathsf{lpo}_1(\mathsf{V}(x), t) & = & \mathsf{false} \\
\mathsf{alpha}_1 & : & \mathsf{Termlist}\ \mathsf{Term} \to \mathsf{Bool} \\
\mathsf{alpha}_1([\,], t) & = & \mathsf{false} \\
\mathsf{alpha}_1(s \centerdot ss, t) & = & s =_{\mathsf{t}} t \vee \mathsf{lpo}_1(s, t) \vee \mathsf{alpha}_1(ss, t) \\
\mathsf{beta}_1 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Bool} \\
\mathsf{beta}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & f >_{\mathsf{F}} g \wedge \mathsf{majo}_1(\mathsf{F}(f, ss), ts) \\
\mathsf{gamma}_1 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Bool} \\
\mathsf{gamma}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & f = g \wedge \mathsf{lex}_1(ss, ts) \wedge \mathsf{majo}_1(\mathsf{F}(f, ss), ts) \\
\mathsf{delta}_1 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Bool} \\
\mathsf{delta}_1(\mathsf{F}(f, ss), \mathsf{V}(y)) & = & \mathsf{contains}_{\mathsf{tl}}(ss, y) \\
\mathsf{majo}_1 & : & \mathsf{Term}\ \mathsf{Termlist} \to \mathsf{Bool} \\
\mathsf{majo}_1(s, [\,]) & = & \mathsf{true} \\
\mathsf{majo}_1(s, t \centerdot ts) & = & \mathsf{lpo}_1(s, t) \wedge \mathsf{majo}_1(s, ts) \\
\mathsf{lex}_1 & : & \mathsf{Termlist}\ \mathsf{Termlist} \to \mathsf{Bool} \\
\mathsf{lex}_1([\,], [\,]) & = & \mathsf{false} \\
\mathsf{lex}_1(s \centerdot ss, t \centerdot ts) & = & \mathbf{if}\ s =_{\mathsf{t}} t\ \mathbf{then}\ \mathsf{lex}_1(ss, ts)\ \mathbf{else}\ \mathsf{lpo}_1(s, t)
\end{array}
$$

The functions beta_1, gamma_1, and delta_1 are not strictly necessary; they are introduced for the sake of clarity. This is in contrast to the function alpha_1, which calls itself tail-recursively. The function majo_1 implements the common part of the cases (β) and (γ) in Definition 4.1, namely $s \succ_{\mathrm{lpo}} t_k$ for all $k \in \{1, \ldots, m\}$. The lexicographic comparison needed by case (γ) is provided by function lex_1, which is only defined for lists of the same length. This partiality reflects that LPO requires function symbols of fixed arity. Therefore, lpo_1 is only defined for well-formed terms (i. e., arguments fulfilling the well predicate).

THEOREM 4.1 *Let s and t be well-formed. Then* $\mathsf{lpo}_1(s, t) = \mathsf{true}$ *iff* $s \succ_{\mathrm{lpo}} t$. □

First optimizations

To get used to the transformational style of program development, we start with some simple optimizations. A quick analysis reveals that it is not advisable to evaluate the case (α) before the cases (β) or (γ). The intuitive explanation is that case (α) reduces the size of the left argument, while the right argument remains the same. This seems disadvantageous for showing the left argument to be greater than the right argument.

[1] The subscript 1 denotes that this is the first version of the implementation. It will increase subsequently.

Hence, reducing the size of the left argument should be delayed.[2] Our second version therefore swaps the order of the cases (α), (β), and (γ):

$$\begin{array}{rcl} \mathsf{lpo}_2 & : & \mathsf{Term\ Term} \rightarrow \mathsf{Bool} \\ \mathsf{lpo}_2(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \mathsf{beta}_2(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \vee \mathsf{gamma}_2(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \\ & & \vee\ \mathsf{alpha}_2(ss, \mathsf{F}(g, ts)) \\ \mathsf{lpo}_2(\mathsf{F}(f, ss), \mathsf{V}(y)) & = & \mathsf{delta}_2(\mathsf{F}(f, ss), \mathsf{V}(y)) \\ \mathsf{lpo}_2(\mathsf{V}(x), t) & = & \mathsf{false} \end{array}$$

The functions alpha_2, ..., are identical to alpha_1, ..., except that they call the $(\cdot)_2$-versions, in particular lpo_2.

THEOREM 4.2 *If $s, t \in \mathsf{Term}$ are well-formed then $\mathsf{lpo}_1(s, t) = \mathsf{lpo}_2(s, t)$.*

PROOF As under these conditions all functions are defined for their arguments this is a consequence of the AC-property of $\vee$. □

A second observation concerns case (γ). If there is some i such that $s_i \succ_{\mathrm{lpo}} t_i$ and $s_j \equiv t_j$ for all $j < i$ the comparisons $s \succ_{\mathrm{lpo}} t_k$ for $1 \leq k \leq i$ are superfluous. This is a corollary of the following lemma.

LEMMA 4.1 *If $s \succ_{\mathrm{st}} s'$ and $s' \succeq_{\mathrm{lpo}} t$ then $s \succ_{\mathrm{lpo}} t$.*

PROOF The LPO contains the subterm relation and is transitive. □

To take advantage of this observation we modify the function gamma to use the newly defined function lexM which combines the lexicographic comparison with the call to majo: $\mathsf{lexM}(s, ss, ts) = \mathsf{lex}(ss, ts) \wedge \mathsf{majo}(s, ts)$ for $s =_{\mathsf{t}} \mathsf{F}(f, ss)$. We get:

$$\begin{array}{rcl} \mathsf{gamma}_3 & : & \mathsf{Term\ Term} \rightarrow \mathsf{Bool} \\ \mathsf{gamma}_3(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & f = g \wedge \mathsf{lexM}_3(\mathsf{F}(f, ss), ss, ts) \\ \mathsf{lexM}_3 & : & \mathsf{Term\ Termlist\ Termlist} \rightarrow \mathsf{Bool} \\ \mathsf{lexM}_3(s, [\,], [\,]) & = & \mathsf{false} \\ \mathsf{lexM}_3(s, s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{if } s_i =_{\mathsf{t}} t_i \textbf{ then } \mathsf{lexM}_3(s, ss, ts) \\ & & \textbf{else } \mathsf{lpo}_3(s_i, t_i) \wedge \mathsf{majo}_3(s, ts) \end{array}$$

It is easy to see that this modification preserves correctness.

THEOREM 4.3 *If $s, t \in \mathsf{Term}$ are well-formed then $\mathsf{lpo}_2(s, t) = \mathsf{lpo}_3(s, t)$.* □

Note that this optimization transfers *positive knowledge* along the branches of the computation: We can avoid some computations because we already know that some ordering relations hold. This is suitable in optimizing conjunctions of conditions.

---

[2] Of course, case (γ) also reduces the size of the left argument, but, in contrast to case (α), it reduces in addition the size of the right argument.

Table 4.1: Time (in seconds) needed for ordering comparisons

problem	number of calls to ordering	time needed by ordering			time needed by other operations
		LPO$_1$	LPO$_2$	LPO$_3$	
GRP180-1	161 446	599.079	171.812	159.734	0.886
LAT020-1	829 439	7.383	4.988	4.137	12.396
LCL109-6	141 266	4.665	2.442	1.751	0.972
RNG027-5	148 787	229.829	110.238	94.140	2.384
z22	2 249	8.728	0.348	0.251	0.066
TPTP$_{10}$	113 105 278	1 158.900	915.300	773.900	1 281.000

First comparisons

A performance analysis shows that the two optimizations have a profound effect on the running time. The three versions of LPO are translated into C functions in a straightforward way[3] and are integrated into the theorem prover WALDMEISTER. We use LPO$_1$, LPO$_2$, and LPO$_3$ to refer to these implementations. As test examples we choose proof tasks with different characteristics. The first four examples are taken from TPTP [SS98]. The example z22 is an encoding of a string-rewriting system as a term-rewriting system. Therefore, there are only unary function symbols and the terms are very deep and narrow with a variable at the leaf. Such examples seem to be absent in TPTP.

Furthermore, we make measurements for all 722 UEQ-problems of TPTP-2.7.0. For that, we first run WALDMEISTER using LPO$_1$ with a time limit of 10 seconds and record how often the ordering routines are called. We then perform the real measurements by running the prover with the different versions LPO$_i$ and by aborting the run after the corresponding number of calls is reached. We use TPTP$_{10}$ to refer to the summarized results.

Table 4.1 contains running times measured on a machine with 1 GHz Pentium III and 4 GByte RAM. For each example we give the number of calls the prover makes to the ordering routines. The next three columns contain the time spent in deciding the ordering relations. The last column gives the time needed by the system *without* the time needed for the orderings.[4] Note the diversity of the examples which can be seen for example in the large differences of the average time needed for one call to the ordering. The running times document that especially the first optimization leads to substantial improvements over the reference implementation. Nevertheless, in most cases the time spent in the ordering is considerably larger than the time for the rest of the system. TPTP$_{10}$ shows different results for two reasons. First, the start overhead

[3] The tail-recursions used in some functions, such as in `alpha`, `lex`, etc., are turned into `while`-loops.

[4] This was determined by running the prover twice. In the first run each result of a call to the ordering was written to a file. In the second run instead of calling the ordering procedure the value of the protocol file was returned. By careful low-level coding we could achieve this without any noticeable overhead, as profiles show.

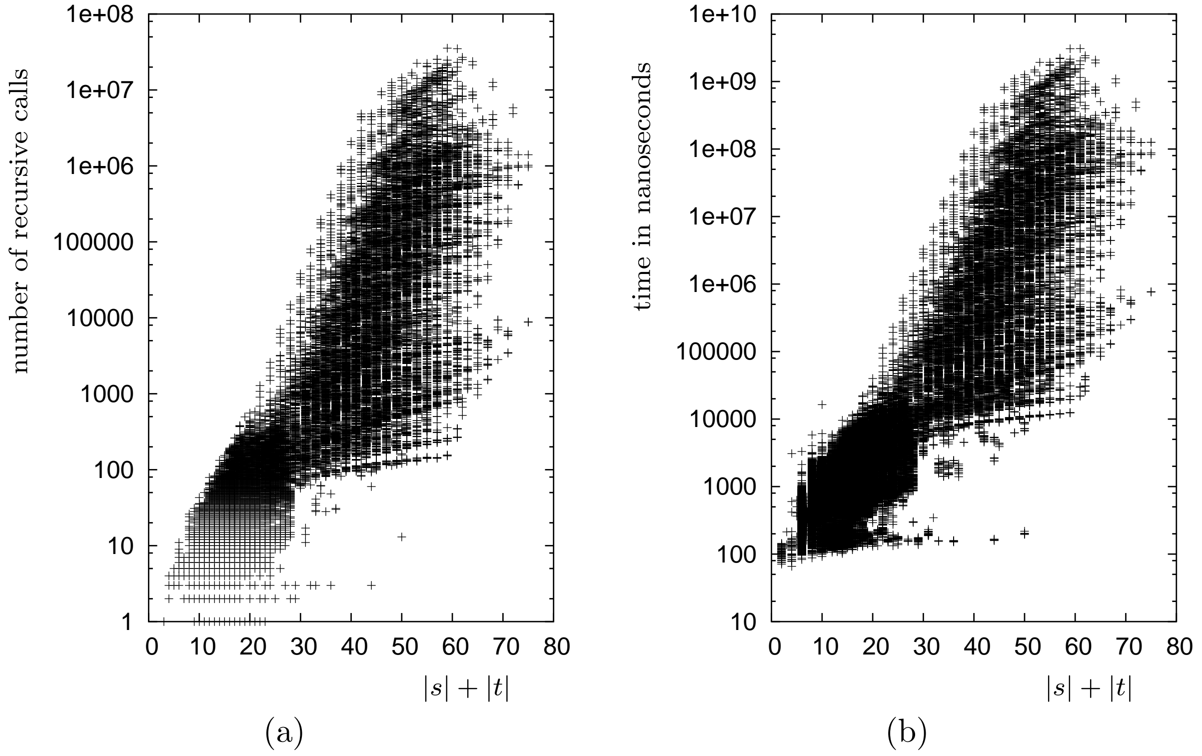


Figure 4.1: Distribution of (a) the number of recursive calls and (b) the time needed to evaluate $\text{LPO}_3(s,t)$ depicted over $|s|+|t|$ for GRP180-1.

for initializing the prover is emphasized by running at most 10 seconds, which shows in the last column. Second, the size of terms tends to increase during the completion, hence larger input sizes occur more rarely in TPTP_{10} than in completed runs. This reduces the differences observed between the variants.

Why does the prover spend so much time in LPO? Fine-grained measurements reveal the following: A single call to the ordering may require a number of recursive calls that is exponential in the size of the input. This can be seen in Figure 4.1 (a). The distribution of running times of single calls reflects the pattern, cf. Figure 4.1 (b). In the proof of the following theorem, we show the reason for this behavior, and that the behavior is independent of the used term data structure.

THEOREM 4.4 *The running time of* $\mathsf{lpo}_i(s,t)$, $i = 1,2,3$, *is in* $\Omega(4^k/\sqrt{k})$ *where* $k = (|s|+|t|)/2$.

PROOF Consider two constants $a, b \in \mathcal{F}$ and two unary function symbols $f, g \in \mathcal{F}$ with precedence $a >_\mathcal{F} b >_\mathcal{F} f >_\mathcal{F} g$. Let $u_n \equiv f^n(b)$ and $v_m \equiv g^m(a)$ Then the variants lpo_1, lpo_2, and lpo_3 all show exponential behavior for the test whether $u_n \succ^?_{\text{lpo}} v_n$. As $v_n \succ_{\text{lpo}} u_n$ holds, all recursive calls return false. Therefore, in each invocation the cases (α) and (β) are both considered. The situation for $n = 3$ is illustrated[5] in Figure 4.2.

[5] Strictly speaking, we have to consider $s \succeq^?_{\text{lpo}} t$ in the recursive calls via case (α). But this reduces essentially to $s \succ^?_{\text{lpo}} t$, because $s \not\equiv t$ can be determined immediately.

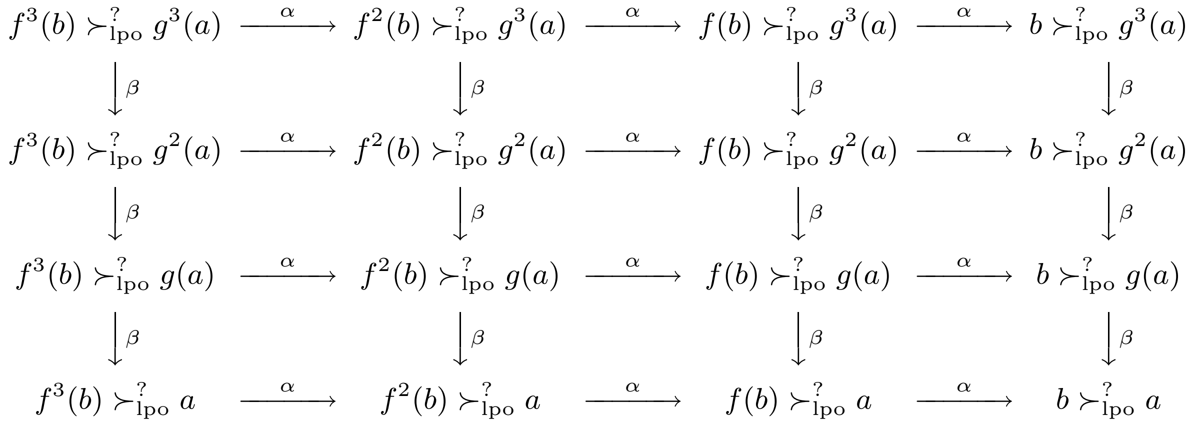


Figure 4.2: Calling graph for test of $f^3(b) \succ^?_{\text{lpo}} g^3(a)$.

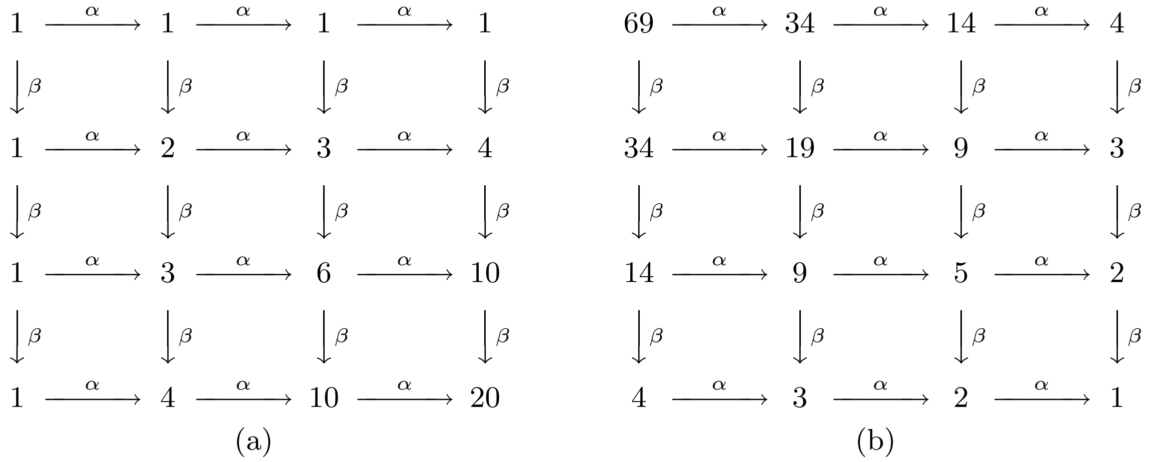


Figure 4.3: (a) Number of invocations of subproblems during the test $f^3(b) \succ^?_{\text{lpo}} g^3(a)$.
(b) Costs $T(n, m)$ associated with each subproblem of $f^3(b) \succ^?_{\text{lpo}} g^3(a)$.

This explains the reason for the exponential behavior: For $u_{n+1} \succ^?_{\text{lpo}} v_{m+1}$ the subproblem $u_n \succ^?_{\text{lpo}} v_m$ is considered twice. Figure 4.3 (a) shows how often each subproblem is considered starting from $n = m = 3$. Obviously, this is the top part of Pascal's triangle of the binomial coefficients.

To determine the costs $T(n, m)$ of testing $u_n \succ^?_{\text{lpo}} v_m$ we count the number of recursive calls. For $n, m \in \{0, \ldots, 3\}$ the values of $T(n, m)$ are depicted in Figure 4.3 (b). We get the following recurrences:

$$
\begin{aligned}
T(n, 0) &= n + 1 \\
T(0, m) &= m + 1 \\
T(n + 1, m + 1) &= 1 + T(n + 1, m) + T(n, m + 1)
\end{aligned}
$$

Here too, we have a connection to the binomial coefficients:

$$T(n,m) \;=\; \binom{n+m+2}{m+1} - 1$$

This can be shown via induction on $n+m$:

$$\begin{aligned}
T(n,0) &= n+1 \;=\; \binom{n+2}{1} - 1 \\
T(0,m) &= m+1 \;=\; \binom{m+2}{m+1} - 1 \\
T(n+1,m+1) &= 1 + T(n+1,m) + T(n,m+1) \\
&= 1 + \binom{(n+1)+m+2}{m+1} - 1 + \binom{n+(m+1)+2}{(m+1)+1} - 1 \\
&= \binom{(n+1)+(m+1)+2}{(m+1)+1} - 1
\end{aligned}$$

Using Stirling's formula we get the asymptotic expression

$$T(n,n) \approx \frac{4^{n+1}}{\sqrt{\pi(n+1)}}$$

which clearly describes an exponential growth. As $|u_n| = |v_n| = n+1$ we can substitute k for $n+1$ and get the desired result. □

For such situations there are two standard techniques to avoid the exponential behavior. The first is *dynamic programming* [Bel57]. Here a table is introduced which contains one entry for each subproblem. The top-down recursion is replaced by a bottom-up computation, which fills each table entry with the help of already determined ones. In our case, we assign in a leftmost-outermost traversal each subterm of a term t a unique number from $1, \ldots, |t|$. Hence, a table of size $|s| \cdot |t|$ is sufficient for $\mathsf{lpo}(s,t)$. For the corresponding subterms s' and t', each entry contains the ordering relations (i. e., whether $s' \equiv t'$, $s' \succ_{\text{lpo}} t'$, $t' \succ_{\text{lpo}} s'$, or s' and t' are incomparable). For computing the entries from large indices downwards to small indices, the work for each entry is determined by the arity of the leading function symbols – the ordering relationships between subterms are already known. This leads to an algorithm with running time in $\Theta(|s| \cdot |t|)$. But the structure of the algorithm is changed in a nontrivial way.

For the second technique, *memoization* [Mic68], the top-down recursion is retained. The result of each function call is stored in a so-called memo-table. When the function is called again with the same arguments, the value from the table is returned and so the recomputation is avoided. This can be thought of as using the table of dynamic programming, but filling in the values of the table lazily. A more typical implementation is with a hash table. The worst-case running time of the resulting algorithm is $O(|s| \cdot |t|)$, recomputations do not occur. If only a fraction of the $|s| \cdot |t|$ subproblems

is needed, this method is faster than dynamic programming. If (nearly) all subproblems are considered, dynamic programming is probably faster; the code is simpler and avoids the overhead of hashing, especially the equality tests for the arguments.

In the literature (e. g. [KNS85, Sny93, Ste94]) the use of dynamic programming is recommended to get a polynomial algorithm for LPO and related orderings. We think its use is appropriate if the ordering relation between all subterms is needed, such as in our test for the unsatisfiability of ordering constraints (see Section 5.4). For the basic ordering comparison, however, our aim is to achieve the same worst-case running time without the use of additional data structures, thus avoiding the work that is associated with allocating and initializing additional memory.

A polynomial version

As a preparation step we first replace the conjunctions in beta and gamma by **if**-expressions using the comparisons of function symbols as guards:

$$\begin{aligned}\mathsf{beta}(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) &= \mathbf{if}\ f >_{\mathsf{F}} g\ \mathbf{then}\ \mathsf{majo}(\mathsf{F}(f,ss),ts)\ \mathbf{else}\ \mathsf{false}\\ \mathsf{gamma}(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) &= \mathbf{if}\ f = g\ \mathbf{then}\ \mathsf{lexM}(\mathsf{F}(f,ss),ss,ts)\ \mathbf{else}\ \mathsf{false}\end{aligned}$$

Then they are inlined into lpo and combined into one **if**-**elif**-expression as the guards are mutually exclusive:

$$\mathsf{lpo}(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) = \begin{pmatrix}\mathbf{if} & f >_{\mathsf{F}} g\ \mathbf{then}\ \mathsf{majo}(\mathsf{F}(f,ss),ts)\\ \mathbf{elif} & f = g\ \ \mathbf{then}\ \mathsf{lexM}(\mathsf{F}(f,ss),ss,ts)\\ \mathbf{else}\ \mathsf{false} & \end{pmatrix} \vee \mathsf{alpha}(ss,\mathsf{F}(g,ts))$$

Finally, we distribute alpha into the different branches and simplify:

$$\begin{aligned}\mathsf{lpo}(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) = {} & \mathbf{if}\ \ f >_{\mathsf{F}} g\ \mathbf{then}\ \mathsf{majo}(\mathsf{F}(f,ss),ts) \vee \mathsf{alpha}(ss,\mathsf{F}(g,ts))\\ & \mathbf{elif}\ f = g\ \ \mathbf{then}\ \mathsf{lexM}(\mathsf{F}(f,ss),ss,ts) \vee \mathsf{alpha}(ss,\mathsf{F}(g,ts))\\ & \mathbf{else}\ \mathsf{alpha}(ss,\mathsf{F}(g,ts))\end{aligned}$$

The performed transformations are typical for an optimizing compiler and do not change the time consumption of the implementation significantly.

The first insight leading to real improvements is that if $\mathsf{majo}(\mathsf{F}(f,ss),ts) = \mathsf{false}$, then $\mathsf{alpha}(ss,\mathsf{F}(g,ts)) = \mathsf{false}$ as well. This means that in contrast to the previous optimizations we can make use of *negative* results from recursive calls which is appropriate for optimizing disjunctions. The optimization is justified by the following lemma, which is formulated positively (i. e., as contraposition of the observation):

LEMMA 4.2 *Let* $s \equiv f(s_1,\ldots,s_n)$ *and* $t \equiv g(t_1,\ldots,t_m)$*. If there is some* $i \in \{1,\ldots,n\}$ *such that* $s_i \succeq_{\text{lpo}} t$ *then* $s \succ_{\text{lpo}} t_j$ *for all* $j \in \{1,\ldots,m\}$*.*

PROOF Because LPO contains the subterm relation, this is a consequence of transitivity. □

This enables us to omit the call to alpha in the first branch of the **if**-expression. For the second branch we combine lexM and alpha into lexMA for $s =_{\mathsf{t}} \mathsf{F}(f,ss)$ and $t =_{\mathsf{t}} \mathsf{F}(f,ts)$:

$$\mathsf{lexMA}(s,t,ss,ts) \;=\; \mathsf{lexM}(s,ss,ts) \lor \mathsf{alpha}(ss,t)$$

Performing the alpha-test is useless for subterms shown equal by the lexicographic test:

LEMMA 4.3 *Let $s \equiv f(s_1,\ldots,s_n)$ and $t \equiv f(t_1,\ldots,t_n)$. If $s_i \equiv t_i$ for some $i \in \{1,\ldots,n\}$, then $s_i \not\succeq_{\text{lpo}} t$.*

PROOF Clear, as we have $t \succ_{\text{lpo}} t_i \equiv s_i$. □

This leads to the following version of lexMA:

$$\begin{array}{rcl} \mathsf{lexMA}(s,t,[\,],[\,]) & = & \mathsf{false} \\ \mathsf{lexMA}(s,t,s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{if } s_i =_{\mathsf{t}} t_i \textbf{ then } \mathsf{lexMA}(s,t,ss,ts) \\ & & \textbf{else } (\mathsf{lpo}(s_i,t_i) \land \mathsf{majo}(s,ts)) \lor \mathsf{alpha}(s_i \centerdot ss, t) \end{array}$$

We turn the conjunction into an **if**-expression and distribute the call to alpha into both branches. After some small simplifications we get:

$$\begin{array}{rcl} \mathsf{lexMA}(s,t,s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{if } \quad s_i =_{\mathsf{t}} t_i \quad \textbf{then } \mathsf{lexMA}(s,t,ss,ts) \\ & & \textbf{elif } \mathsf{lpo}(s_i,t_i) \textbf{ then } \mathsf{majo}(s,ts) \lor \mathsf{alpha}(s_i \centerdot ss,t) \\ & & \textbf{else } \mathsf{alpha}(s_i \centerdot ss, t) \end{array}$$

Now, two further optimizations are possible. Using Lemma 4.2 the first call to alpha can be completely eliminated. For the second call to alpha we already know from the evaluation of the guards that $s_i \not\succeq_{\text{lpo}} t_i$. Hence $s_i \not\succeq_{\text{lpo}} t$, and s_i can be dropped from the argument list.

Summing up, we get the following version of LPO:

$$\begin{array}{rcl} \mathsf{lpo}_4 & : & \mathsf{Term\ Term} \to \mathsf{Bool} \\ \mathsf{lpo}_4(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) & = & \textbf{if } \quad f >_{\mathsf{F}} g \textbf{ then } \mathsf{majo}_4(\mathsf{F}(f,ss),ts) \\ & & \textbf{elif } \; f = g \quad \textbf{then } \mathsf{lexMA}_4(\mathsf{F}(f,ss),\mathsf{F}(g,ts),ss,ts) \\ & & \textbf{else } \mathsf{alpha}_4(ss,\mathsf{F}(g,ts)) \\ \mathsf{lpo}_4(\mathsf{F}(f,ss),\mathsf{V}(y)) & = & \mathsf{delta}_4(\mathsf{F}(f,ss),\mathsf{V}(y)) \\ \mathsf{lpo}_4(\mathsf{V}(x),t) & = & \mathsf{false} \\ \mathsf{lexMA}_4 & : & \mathsf{Term\ Term\ Termlist\ Termlist} \to \mathsf{Bool} \\ \mathsf{lexMA}_4(s,t,[\,],[\,]) & = & \mathsf{false} \\ \mathsf{lexMA}_4(s,t,s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{if } \quad s_i =_{\mathsf{t}} t_i \quad \textbf{then } \mathsf{lexMA}_4(s,t,ss,ts) \\ & & \textbf{elif } \mathsf{lpo}_4(s_i,t_i) \textbf{ then } \mathsf{majo}_4(s,ts) \\ & & \textbf{else } \mathsf{alpha}_4(ss,t) \end{array}$$

Note that no pair of subterms is compared twice.

COROLLARY 4.1 *If $s,t \in \mathsf{Term}$ are well-formed then $\mathsf{lpo}_3(s,t) = \mathsf{lpo}_4(s,t)$.* □

It remains to show that this version needs time that is polynomially bound in the size of the arguments.

LEMMA 4.4 *Evaluating* $\mathsf{lpo}_4(s,t)$ *needs* $|s| \cdot |t|$ *recursive calls in the worst case.*

PROOF Induction on $|s| + |t|$. Additional recursive calls occur only if both terms start with a function symbol. Let $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv g(t_1, \ldots, t_m)$. We proceed by case analysis.

(i) $f >_{\mathcal{F}} g$. Then s will be compared with each t_j, $j \in \{1, \ldots, m\}$. By induction hypothesis, each of them will need $|s| \cdot |t_j|$ recursive calls in the worst case, which gives

$$1 + \sum_{j=1}^{m} |s| \cdot |t_j| \le |s| \cdot |t| \ .$$

(ii) $f = g$. If $s \equiv t$, then no additional recursive call is performed. Otherwise, there is some i with $s_i \not\equiv t_i$. If $s_i \succ_{\text{lpo}} t_i$, then s is compared against t_k for $k > i$. By induction hypothesis we get

$$1 + |s_i| \cdot |t_i| + \sum_{k=i+1}^{n} |s| \cdot |t_k| < 1 + \sum_{j=1}^{n} |s| \cdot |t_j| \le |s| \cdot |t| \ .$$

In the other case, we have $s_i \not\succ_{\text{lpo}} t_i$ and the remaining subterms of s are compared to t. By induction hypothesis we have

$$1 + |s_i| \cdot |t_i| + \sum_{k=i+1}^{n} |s_k| \cdot |t| < 1 + \sum_{j=1}^{n} |s_j| \cdot |t| \le |s| \cdot |t| \ .$$

(iii) $f \not\geq_{\mathcal{F}} g$. In the worst case each subterm s_j of s is compared to t. Applying induction hypothesis leads to

$$1 + \sum_{j=1}^{n} |s_j| \cdot |t| \le |s| \cdot |t| \ .$$

□

For determining the costs per recursive call, we have to distinguish whether the top-symbols of the arguments are both function symbols or not. In the first case, the work consists mainly of the comparison of the function symbols with precedence $>_{\mathsf{F}}$, which we can assume to need constant time. In addition, some functions (alpha_4 and lexMA_4) compare subterms for syntactic equality, which can be performed in linear time in the size of the arguments. In the second case, either constant time is needed to return false or $\mathsf{contains}_{\mathsf{tl}}$ is called which needs time that is linear in the size of the first argument. This leads to the desired result:

THEOREM 4.5 *Evaluating* $\mathsf{lpo}_4(s,t)$ *needs polynomial time in the worst case.* □

Combining the ordering test with syntactic equality

During the evaluation of $\mathsf{lpo}_4(s,t)$ some subterms may be considered both for syntactic equality and for the ordering relation between them. Hence, the symbols in these

subterms may be considered twice. We can improve on this by combining the ordering test with the test for syntactic equality. The resulting function can return three values as one of $s \equiv t$, $s \succ_{\text{lpo}} t$, or $s \not\succeq_{\text{lpo}} t$ holds. We therefore introduce a new sort Res with constructors $\mathsf{E}, \mathsf{G}, \mathsf{N} :\rightarrow \mathsf{Res}$. We then use the following wrapper shown on the left-hand side to use the newly defined $\mathsf{lpo_R}$ shown on the right-hand side:

$$\begin{array}{rcl} \mathsf{lpo}_5 & : & \mathsf{Term\ Term} \rightarrow \mathsf{Bool} \\ \mathsf{lpo}_5(s,t) & = & \textbf{case } \mathsf{lpo_R}(s,t) \\ & & \quad \mathsf{G} \quad : \mathsf{true} \\ & & \quad \mathsf{E}, \mathsf{N} : \mathsf{false} \end{array} \qquad \begin{array}{rcl} \mathsf{lpo_R} & : & \mathsf{Term\ Term} \rightarrow \mathsf{Res} \\ \mathsf{lpo_R}(s,t) & = & \textbf{if} \quad s =_{\mathsf{t}} t \quad \textbf{then } \mathsf{E} \\ & & \textbf{elif } \mathsf{lpo}_4(s,t) \textbf{ then } \mathsf{G} \\ & & \textbf{else } \mathsf{N} \end{array}$$

To transform $\mathsf{lpo_R}$ we start with some case splitting. If one or both arguments are variables, it is sufficient to unfold the definitions and to do some simplifications:

$$\begin{array}{rcl} \mathsf{lpo_R}(\mathsf{V}(x), \mathsf{V}(y)) & = & \textbf{if } x = y \textbf{ then } \mathsf{E} \textbf{ else } \mathsf{N} \\ \mathsf{lpo_R}(\mathsf{F}(f, ss), \mathsf{V}(y)) & = & \textbf{if } \mathsf{contains_{tl}}(ss, y) \textbf{ then } \mathsf{G} \textbf{ else } \mathsf{N} \\ \mathsf{lpo_R}(\mathsf{V}(x), \mathsf{F}(g, ts)) & = & \mathsf{N} \end{array}$$

For the remaining case we also unfold the definitions:

$$\begin{array}{l} \mathsf{lpo_R}(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \; = \\ \quad \textbf{if} \quad f = g \wedge ss =_{\mathsf{tl}} ts \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathsf{E} \\ \quad \textbf{elif} \left(\begin{array}{l} \textbf{if} \quad f >_{\mathsf{F}} g \textbf{ then } \mathsf{majo}(\mathsf{F}(f, ss), ts) \\ \textbf{elif } f = g \quad \textbf{then } \mathsf{lexMA}(\mathsf{F}(f, ss), \mathsf{F}(g, ts), ss, ts) \\ \textbf{else } \mathsf{alpha}(ss, \mathsf{F}(g, ts)) \end{array} \right) \textbf{then } \mathsf{G} \\ \quad \textbf{else } \mathsf{N} \end{array}$$

Then we reorganize the nested **if**-expressions by using the comparisons of the leading function symbols as the main guards:

$$\begin{array}{l} \mathsf{lpo_R}(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \; = \\ \quad \textbf{if} \quad f >_{\mathsf{F}} g \textbf{ then if} \quad \mathsf{majo}(\mathsf{F}(f, ss), ts) \textbf{ then } \mathsf{G} \\ \qquad\qquad\qquad\qquad \textbf{else } \mathsf{N} \\ \quad \textbf{elif } f = g \quad \textbf{then if} \quad ss =_{\mathsf{tl}} ts \qquad\qquad\qquad\qquad \textbf{then } \mathsf{E} \\ \qquad\qquad\qquad\qquad \textbf{elif } \mathsf{lexMA}(\mathsf{F}(f, ss), \mathsf{F}(g, ts), ss, ts) \textbf{ then } \mathsf{G} \\ \qquad\qquad\qquad\qquad \textbf{else } \mathsf{N} \\ \quad \textbf{else if} \quad \mathsf{alpha}(ss, \mathsf{F}(g, ts)) \textbf{ then } \mathsf{G} \\ \qquad\quad \textbf{else } \mathsf{N} \end{array}$$

To get a structure that is similar to lpo_4 we have to combine the test for syntactic equality and lexMA into one function which we call lexMAE. Furthermore, we have to modify majo and alpha such that they are Res-valued. This results in the following version of $\mathsf{lpo_R}$:

$$\begin{array}{rcl} \mathsf{lpo_R}(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \textbf{if} \quad f >_{\mathsf{F}} g \textbf{ then } \mathsf{majo_R}(s, ts) \\ & & \textbf{elif } f = g \quad \textbf{then } \mathsf{lexMAE}(\mathsf{F}(f, ss), \mathsf{F}(g, ts), ss, ts) \\ & & \textbf{else } \mathsf{alpha_R}(ss, t) \end{array}$$

For the remaining functions we start with the following specifications:

$$\begin{array}{rcl}
\mathsf{lexMAE} & : & \mathsf{Term\ Term\ Termlist\ Termlist} \rightarrow \mathsf{Res} \\
\mathsf{lexMAE}(s,t,ss,ts) & = & \textbf{if}\ \ ss =_{\mathsf{tl}} ts \ \ \textbf{then}\ \mathsf{E} \\
 & & \textbf{elif}\ \ \mathsf{lexMA}(s,t,ss,ts)\ \textbf{then}\ \mathsf{G} \\
 & & \textbf{else}\ \mathsf{N} \\
\mathsf{majo_R} & : & \mathsf{Term\ Termlist} \rightarrow \mathsf{Res} \\
\mathsf{majo_R}(s,ts) & = & \textbf{if}\ \ \mathsf{majo}(s,ts)\ \textbf{then}\ \mathsf{G} \\
 & & \textbf{else}\ \mathsf{N} \\
\mathsf{alpha_R} & : & \mathsf{Termlist\ Term} \rightarrow \mathsf{Res} \\
\mathsf{alpha_R}(ss,t) & = & \textbf{if}\ \ \mathsf{alpha}(ss,t)\ \textbf{then}\ \mathsf{G} \\
 & & \textbf{else}\ \mathsf{N}
\end{array}$$

After case splitting, unfolding of definitions, and some simplification function lexMAE has the following form:

$$\begin{array}{rcl}
\mathsf{lexMAE}(s,t,[\,],[\,]) & = & \mathsf{E} \\
\mathsf{lexMAE}(s,t,s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{if}\ \ s_i =_{\mathsf{t}} t_i \wedge ss =_{\mathsf{tl}} ts \qquad \textbf{then}\ \mathsf{E} \\
 & & \textbf{elif}\ \left(\begin{array}{l}\textbf{if}\ \ s_i =_{\mathsf{t}} t_i \ \ \textbf{then}\ \mathsf{lexMA}(s,t,ss,ts) \\ \textbf{elif}\ \mathsf{lpo_4}(s_i,t_i)\ \textbf{then}\ \mathsf{majo}(s,ts) \\ \textbf{else}\ \mathsf{alpha}(ss,t)\end{array}\right)\ \textbf{then}\ \mathsf{G} \\
 & & \textbf{else}\ \mathsf{N}
\end{array}$$

Reorganizing the guards leads to

$$\begin{array}{rcl}
\mathsf{lexMAE}(s,t,s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{if}\ \ s_i =_{\mathsf{t}} t_i \ \ \textbf{then}\ \begin{array}[t]{l}\textbf{if}\ \ ss =_{\mathsf{tl}} ts \ \ \textbf{then}\ \mathsf{E} \\ \textbf{elif}\ \ \mathsf{lexMA}(s,t,ss,ts)\ \textbf{then}\ \mathsf{G} \\ \textbf{else}\ \mathsf{N}\end{array} \\
 & & \textbf{elif}\ \ \mathsf{lpo_4}(s_i,t_i)\ \textbf{then}\ \begin{array}[t]{l}\textbf{if}\ \ \mathsf{majo}(s,ts)\ \textbf{then}\ \mathsf{G} \\ \textbf{else}\ \mathsf{N}\end{array} \\
 & & \textbf{else if}\ \ \mathsf{alpha}(ss,t)\ \textbf{then}\ \mathsf{G} \\
 & & \qquad \textbf{else}\ \mathsf{N}
\end{array}$$

Now we can use $\mathsf{lpo_R}$ to avoid the double traversal of s_i and t_i. We therefore have to turn the outer **if**-**elif** construction into a **case**-expression:

$$\begin{array}{rcl}
\mathsf{lexMAE}(s,t,s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{case}\ \mathsf{lpo_R}(s_i,t_i) \\
 & & \quad \mathsf{E} : \begin{array}[t]{l}\textbf{if}\ \ ss =_{\mathsf{tl}} ts \ \ \textbf{then}\ \mathsf{E} \\ \textbf{elif}\ \ \mathsf{lexMA}(s,t,ss,ts)\ \textbf{then}\ \mathsf{G} \\ \textbf{else}\ \mathsf{N}\end{array} \\
 & & \quad \mathsf{G} : \begin{array}[t]{l}\textbf{if}\ \ \mathsf{majo}(s,ts)\ \textbf{then}\ \mathsf{G} \\ \textbf{else}\ \mathsf{N}\end{array} \\
 & & \quad \mathsf{N} : \begin{array}[t]{l}\textbf{if}\ \ \mathsf{alpha}(ss,t)\ \textbf{then}\ \mathsf{G} \\ \textbf{else}\ \mathsf{N}\end{array}
\end{array}$$

Finally, we can fold back the definitions of lexMAE, $\mathsf{majo_R}$, and $\mathsf{alpha_R}$.

Following the same scheme – case splitting, unfolding of definitions, reorganizing the guards, introducing **case**-expressions, and folding back – we can transform $\mathsf{alpha_R}$ and $\mathsf{majo_R}$. We finally get the following definitions of $\mathsf{lpo_{R5}}$:

$$
\begin{array}{rcl}
\mathsf{lpo}_{\mathsf{R5}} & : & \mathsf{Term\ Term} \rightarrow \mathsf{Res} \\
\mathsf{lpo}_{\mathsf{R5}}(\mathsf{V}(x), \mathsf{V}(y)) & = & \textbf{if } x = y \textbf{ then } \mathsf{E} \textbf{ else } \mathsf{N} \\
\mathsf{lpo}_{\mathsf{R5}}(\mathsf{F}(f, ss), \mathsf{V}(y)) & = & \textbf{if } \mathsf{contains}_{\mathsf{tl}}(ss, y) \textbf{ then } \mathsf{G} \textbf{ else } \mathsf{N} \\
\mathsf{lpo}_{\mathsf{R5}}(\mathsf{V}(x), \mathsf{F}(g, ts)) & = & \mathsf{N} \\
\mathsf{lpo}_{\mathsf{R5}}(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \textbf{if } \quad f >_{\mathsf{F}} g \textbf{ then } \mathsf{majo}_{\mathsf{R5}}(\mathsf{F}(f, ss), ts) \\
& & \textbf{elif } \ f = g \quad \textbf{then } \mathsf{lexMAE}_5(\mathsf{F}(f, ss), \mathsf{F}(g, ts), ss, ts) \\
& & \textbf{else } \mathsf{alpha}_{\mathsf{R5}}(ss, t) \\
\mathsf{lexMAE}_5 & : & \mathsf{Term\ Term\ Termlist\ Termlist} \rightarrow \mathsf{Res} \\
\mathsf{lexMAE}_5(s, t, [\,], [\,]) & = & \mathsf{E} \\
\mathsf{lexMAE}_5(s, t, s_i \centerdot ss, t_i \centerdot ts) & = & \textbf{case } \mathsf{lpo}_{\mathsf{R5}}(s_i, t_i) \\
& & \quad \mathsf{E} : \mathsf{lexMAE}_5(s, t, ss, ts) \\
& & \quad \mathsf{G} : \mathsf{majo}_{\mathsf{R5}}(s, ts) \\
& & \quad \mathsf{N} : \mathsf{alpha}_{\mathsf{R5}}(ss, t) \\
\mathsf{majo}_{\mathsf{R5}} & : & \mathsf{Term\ Termlist} \rightarrow \mathsf{Res} \\
\mathsf{majo}_{\mathsf{R5}}(s, [\,]) & = & \mathsf{G} \\
\mathsf{majo}_{\mathsf{R5}}(s, t \centerdot ts) & = & \textbf{case } \mathsf{lpo}_{\mathsf{R5}}(s, t) \\
& & \quad \mathsf{G} \quad\ : \mathsf{majo}_{\mathsf{R5}}(s, ts) \\
& & \quad \mathsf{E}, \mathsf{N} : \mathsf{N} \\
\mathsf{alpha}_{\mathsf{R5}} & : & \mathsf{Termlist\ Term} \rightarrow \mathsf{Res} \\
\mathsf{alpha}_{\mathsf{R5}}([\,], t) & = & \mathsf{N} \\
\mathsf{alpha}_{\mathsf{R5}}(s \centerdot ss, t) & = & \textbf{case } \mathsf{lpo}_{\mathsf{R5}}(s, t) \\
& & \quad \mathsf{E}, \mathsf{G} : \mathsf{G} \\
& & \quad \mathsf{N} \quad\ : \mathsf{alpha}_{\mathsf{R5}}(ss, t)
\end{array}
$$

If we compare the transformations used to derive $\mathsf{lpo}_{\mathsf{R5}}$ with those used for the previous optimizations, we see that no further domain-specific knowledge about LPO is needed. The transformations in this section rely solely on some standard Unfold/Fold-calculus [BD77].

LEMMA 4.5 *Let* $s, t \in \mathsf{Term}$ *be well-formed. Then* $\mathsf{lpo}_{\mathsf{R5}}(s, t) = \mathsf{G}$ *iff* $\mathsf{lpo}_4(s, t) = \mathsf{true}$ *and* $\mathsf{lpo}_{\mathsf{R5}}(s, t) = \mathsf{E}$ *iff* $s =_{\mathsf{t}} t$*. The worst-case running time of* $\mathsf{lpo}_{\mathsf{R5}}(s, t)$ *is* $O(|s| \cdot |t|)$.

PROOF The program transformations preserve the semantics of the functions. Hence, $\mathsf{lpo}_{\mathsf{R5}}(s, t) = \mathsf{G}$ iff $\mathsf{lpo}_4(s, t) = \mathsf{true}$ and $\mathsf{lpo}_{\mathsf{R5}}(s, t) = \mathsf{E}$ iff $s =_{\mathsf{t}} t$ by the initial definition of $\mathsf{lpo}_{\mathsf{R5}}$. As we can assume that the comparison of two symbols needs constant time, it suffices to count the number of pairs of symbols that are compared during the evaluation. If at least one argument of $\mathsf{lpo}_{\mathsf{R5}}$ is a variable then the number of comparisons is either one or is linear in the size of the other argument. For the remaining case, we can show analogously to the proof of Lemma 4.4 that during the evaluation of $\mathsf{lpo}_{\mathsf{R5}}(s, t)$ at most $|s| \cdot |t|$ pairs of symbols are considered. In difference to lpo_4, no separate tests occur for syntactic equality of subterms. Hence, the evaluation of $\mathsf{lpo}_{\mathsf{R5}}(s, t)$ is in $O(|s| \cdot |t|)$.

In the following we show that for certain inputs $\mathsf{lpo}_{\mathsf{R5}}(s, t)$ needs time proportional to $|s| \cdot |t|$.

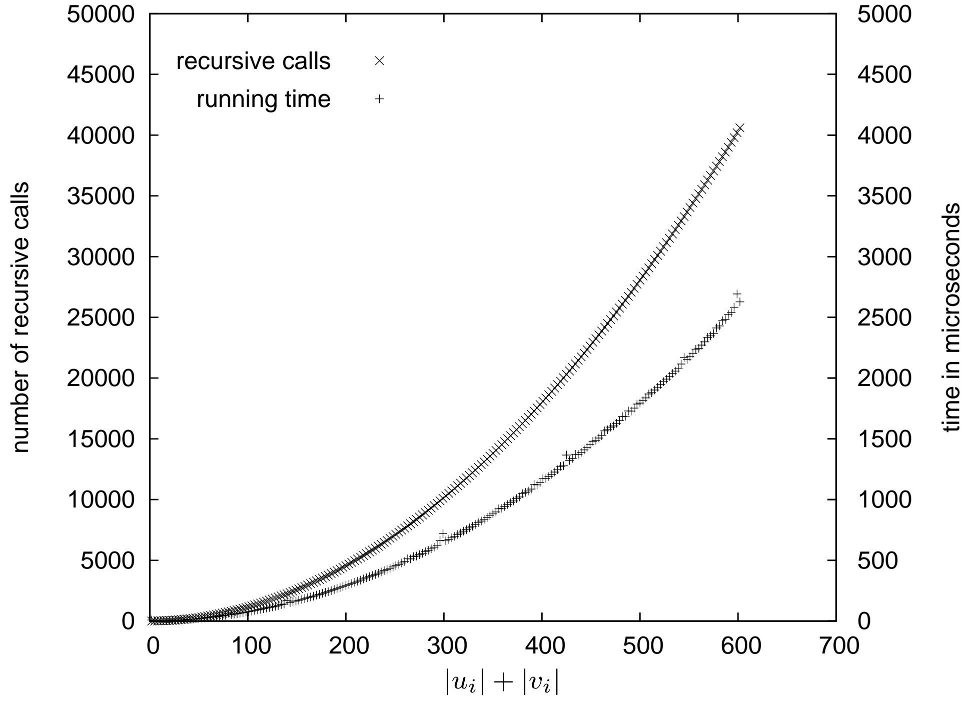


Figure 4.4: Number of recursive calls and running times for $\mathsf{lpo}_{\mathsf{R5}}(u_i, v_i)$, $i = 0, \ldots, 200$, (cf. proof of Lemma 4.6).

Lemma 4.6 *Evaluating* $\mathsf{lpo}_{\mathsf{R5}}(s,t)$ *is in* $\Omega(|s| \cdot |t|)$.

Proof Consider two constants $a, b \in \mathcal{F}$, a unary function symbol $f \in \mathcal{F}$, and a binary one $g \in \mathcal{F}$ with precedence $a >_\mathcal{F} b >_\mathcal{F} f >_\mathcal{F} g$. Then the terms u_i and v_i are recursively defined by $u_0 \equiv a$ and $u_{i+1} \equiv f(u_i)$, respectively $v_0 \equiv b$ and $v_{i+1} \equiv g(v_0, v_i)$. Therefore, $|u_i| = i+1$ and $|v_i| = 2i+1$, and $u_i \not\equiv v_j$ for all $i, j \geq 0$. The evaluation of $\mathsf{lpo}_{\mathsf{R5}}(u_i, v_j)$ makes $(i+1)(j+1)+j$ recursive calls. This can be shown by induction on $i+j$. Let $j = 0$. The comparison of u_0 with v_0 needs one call – both terms have no subterms. As $f \not\geq_\mathcal{F} b$ the call of $\mathsf{lpo}_{\mathsf{R5}}(u_{i+1}, v_0)$ results via $\mathsf{alpha}_{\mathsf{R5}}$ in one call to $\mathsf{lpo}_{\mathsf{R5}}(u_i, v_0)$; the number of recursive calls for $\mathsf{lpo}_{\mathsf{R5}}(u_i, v_0)$ is therefore $i+1$. For the comparison of u_i with $v_{j+1} \equiv g(v_0, v_j)$ two recursive calls to $\mathsf{lpo}_{\mathsf{R5}}(u_i, v_0)$ and $\mathsf{lpo}_{\mathsf{R5}}(u_i, v_j)$ occur, because g is in the precedence smaller than f and than a. This makes $1+(i+1)+(i+1)(j+1)+j+1 = (i+1)(j+2)+j+2$ calls by use of induction hypothesis. □

It can be seen in Figure 4.4 that for the example of the previous proof the measured running times quite well reflect the number of recursive calls, thus giving experimental support to the theoretical analysis.

Theorem 4.6 *The worst-case running time of* $\mathsf{lpo}_{\mathsf{R5}}(s,t)$ *is* $\Theta(|s| \cdot |t|)$. □

In some situations we get a better bound:

LEMMA 4.7 *If the maximal arity of function symbols is one, the worst-case running time of* $\mathsf{lpo}_{\mathsf{R5}}(s,t)$ *is* $O(|s|+|t|)$.

PROOF Depending on the relationship of the top symbols at most one recursive call occurs for which the size of the arguments is one or two symbols smaller. □

Therefore, $\mathsf{lpo}_{\mathsf{R5}}$ can handle the example used in the proof of Theorem 4.4 within linear time.

Bidirectional comparisons

During theorem proving we are often not only interested in determining whether a term is greater than another term with respect to LPO, but in which ordering relation the two terms are. As LPO establishes on terms a partial ordering, we can have for terms s and t either $s \equiv t$, $s \succ_{\mathrm{lpo}} t$, $t \succ_{\mathrm{lpo}} s$, or that s and t are incomparable. Hence, it suffices to extend Res by an additional constructor $\mathsf{L} :\to \mathsf{Res}$ to capture the four cases. Function clpo determines the ordering relationship of two terms. For the first four versions of LPO its definition is depicted on the left-hand side ($i = 1, \ldots, 4$), for $\mathsf{lpo}_{\mathsf{R5}}$ on the right-hand side (as $\mathsf{lpo}_{\mathsf{R5}}$ returns Res and not Bool we have to use a different definition):

$$\begin{array}{rcl} \mathsf{clpo}_i & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Res} \\ \mathsf{clpo}_i(s,t) & = & \textbf{if}\ \ s =_{\mathsf{t}} t\ \ \textbf{then}\ \mathsf{E} \\ & & \textbf{elif}\ \ \mathsf{lpo}_i(s,t)\ \textbf{then}\ \mathsf{G} \\ & & \textbf{elif}\ \ \mathsf{lpo}_i(t,s)\ \textbf{then}\ \mathsf{L} \\ & & \textbf{else}\ \mathsf{N} \end{array} \qquad \begin{array}{rcl} \mathsf{clpo}_5 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Res} \\ \mathsf{clpo}_5(s,t) & = & \textbf{case}\ \mathsf{lpo}_{\mathsf{R5}}(s,t) \\ & & \quad \mathsf{E} : \mathsf{E} \\ & & \quad \mathsf{G} : \mathsf{G} \\ & & \quad \mathsf{N} : \mathsf{flip}(\mathsf{lpo}_{\mathsf{R5}}(t,s)) \end{array}$$

Function $\mathsf{flip} : \mathsf{Res} \to \mathsf{Res}$ is a small function with the following definition:

$$\begin{array}{rclcrcl} \mathsf{flip}(\mathsf{E}) & = & \mathsf{E} & \quad & \mathsf{flip}(\mathsf{L}) & = & \mathsf{G} \\ \mathsf{flip}(\mathsf{N}) & = & \mathsf{N} & & \mathsf{flip}(\mathsf{G}) & = & \mathsf{L} \end{array}$$

With these definitions it may happen that both invocations of lpo_i, $i = 1, \ldots, 4$, or $\mathsf{lpo}_{\mathsf{R5}}$ will traverse both terms comparing the same subterms – only in different directions. Consider the (pathological) example $s \equiv f^n(x)$ and $t \equiv f^n(y)$, which are incomparable. To determine this, function clpo_4 makes $O(n^2)$ steps as in each recursive call of lpo_4 the arguments $f^i(x)$ and $f^i(y)$ are first tested in $i+1$ steps for syntactic equality before invoking $\mathsf{lpo}_4(f^{i-1}(x), f^{i-1}(y))$. Function clpo_5 is asymptotically better, it needs only linear time. However, it is still not optimal. It makes $2(n+1)$ steps because $\mathsf{lpo}_{\mathsf{R5}}$ is called twice. Our aim is to develop a variant that avoids duplicated work. Our transformations will combine the traversals into one in most cases, so the resulting function clpo_6 will make $n+1$ steps for this example. As the transformations in this section are similar to the transformations of the previous section we skip several parts and focus on the most interesting points.

For its symmetric structure we use clpo_4 as a starting point. First, we do some case splitting. The cases where at least one argument is a variable can be handled by unfolding the definitions and simplification. For the remaining case we unfold the

definitions and reorganize the nested **if**-expressions to use the comparisons of the leading function symbols as the main guards. We get:

$$
\begin{array}{l}
\mathsf{clpo}(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) \;=\\
\quad \textbf{if}\ \ f=g\ \ \textbf{then}\ \textbf{if}\ \ ss =_{\mathsf{tl}} ts \qquad\qquad\qquad\qquad \textbf{then}\ \mathsf{E}\\
\qquad\qquad\qquad\qquad\ \textbf{elif}\ \mathsf{lexMA}(\mathsf{F}(f,ss),\mathsf{F}(g,ts),ss,ts)\ \textbf{then}\ \mathsf{G}\\
\qquad\qquad\qquad\qquad\ \textbf{elif}\ \mathsf{lexMA}(\mathsf{F}(g,ts),\mathsf{F}(f,ss),ts,ss)\ \textbf{then}\ \mathsf{L}\\
\qquad\qquad\qquad\qquad\ \textbf{else}\ \mathsf{N}\\
\quad \textbf{elif}\ f >_{\mathsf{F}} g\ \textbf{then}\ \textbf{if}\ \ \mathsf{majo}(\mathsf{F}(f,ss),ts)\ \textbf{then}\ \mathsf{G}\\
\qquad\qquad\qquad\qquad\ \textbf{elif}\ \mathsf{alpha}(ts,\mathsf{F}(f,ss))\ \textbf{then}\ \mathsf{L}\\
\qquad\qquad\qquad\qquad\ \textbf{else}\ \mathsf{N}\\
\quad \textbf{elif}\ g >_{\mathsf{F}} f\ \textbf{then}\ \textbf{if}\ \ \mathsf{majo}(\mathsf{F}(g,ts),ss)\ \textbf{then}\ \mathsf{L}\\
\qquad\qquad\qquad\qquad\ \textbf{elif}\ \mathsf{alpha}(ss,\mathsf{F}(g,ts))\ \textbf{then}\ \mathsf{G}\\
\qquad\qquad\qquad\qquad\ \textbf{else}\ \mathsf{N}\\
\quad \textbf{else}\ \textbf{if}\ \ \mathsf{alpha}(ss,\mathsf{F}(g,ts))\ \textbf{then}\ \mathsf{G}\\
\qquad\quad \textbf{elif}\ \mathsf{alpha}(ts,\mathsf{F}(f,ss))\ \textbf{then}\ \mathsf{L}\\
\qquad\quad \textbf{else}\ \mathsf{N}
\end{array}
$$

As can be seen, the second and the third branch are nearly symmetric – except return values. Because of function flip, it suffices to introduce only three auxiliary functions to get a concise version of clpo:

$$
\begin{array}{rcl}
\mathsf{clpo}(\mathsf{F}(f,ss),\mathsf{F}(g,ts)) & = & \textbf{if}\ \ f=g\ \ \textbf{then}\ \mathsf{cLMA}(\mathsf{F}(f,ss),\mathsf{F}(g,ts),ss,ts)\\
& & \textbf{elif}\ f >_{\mathsf{F}} g\ \textbf{then}\ \mathsf{cMA}(\mathsf{F}(f,ss),ts)\\
& & \textbf{elif}\ g >_{\mathsf{F}} f\ \textbf{then}\ \mathsf{flip}(\mathsf{cMA}(\mathsf{F}(g,ts),ss))\\
& & \textbf{else}\ \mathsf{cAA}(\mathsf{F}(f,ss),\mathsf{F}(g,ts),ss,ts)
\end{array}
$$

Note that the call to cAA is necessary because the precedence is only a partial ordering on $\mathcal{F}$. The initial specifications of the auxiliary functions are as follows:

$$
\begin{array}{rcl}
\mathsf{cMA} & : & \mathsf{Term}\ \mathsf{Termlist} \rightarrow \mathsf{Res}\\
\mathsf{cMA}(s,ts) & = & \textbf{if}\ \ \mathsf{majo}(s,ts)\ \textbf{then}\ \mathsf{G}\\
& & \textbf{elif}\ \mathsf{alpha}(ts,s)\ \textbf{then}\ \mathsf{L}\\
& & \textbf{else}\ \mathsf{N}\\
\mathsf{cLMA} & : & \mathsf{Term}\ \mathsf{Term}\ \mathsf{Termlist}\ \mathsf{Termlist} \rightarrow \mathsf{Res}\\
\mathsf{cLMA}(s,t,ss,ts) & = & \textbf{if}\ \ ss =_{\mathsf{tl}} ts \qquad\quad \textbf{then}\ \mathsf{E}\\
& & \textbf{elif}\ \mathsf{lexMA}(s,t,ss,ts)\ \textbf{then}\ \mathsf{G}\\
& & \textbf{elif}\ \mathsf{lexMA}(t,s,ts,ss)\ \textbf{then}\ \mathsf{L}\\
& & \textbf{else}\ \mathsf{N}\\
\mathsf{cAA} & : & \mathsf{Term}\ \mathsf{Term}\ \mathsf{Termlist}\ \mathsf{Termlist} \rightarrow \mathsf{Res}\\
\mathsf{cAA}(s,t,ss,ts) & = & \textbf{if}\ \ \mathsf{alpha}(ss,t)\ \textbf{then}\ \mathsf{G}\\
& & \textbf{elif}\ \mathsf{alpha}(ts,s)\ \textbf{then}\ \mathsf{L}\\
& & \textbf{else}\ \mathsf{N}
\end{array}
$$

The optimization of the three auxiliary functions is similar to the transformations of the previous section. Hence, we skip them for space reasons. We finally get the following definition of clpo_6. Note that the definitions of $\mathsf{lpo}_{\mathsf{R6}}$, $\mathsf{alpha}_{\mathsf{R6}}$, etc., are identical to their $\mathsf{lpo}_{\mathsf{R5}}$-counterparts.

$$
\begin{array}{rcl}
\mathsf{clpo}_6 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Res} \\
\mathsf{clpo}_6(\mathsf{F}(f, ss), \mathsf{V}(y)) & = & \mathbf{if}\ \mathsf{contains}_{\mathsf{tl}}(ss, y)\ \mathbf{then}\ \mathsf{G}\ \mathbf{else}\ \mathsf{N} \\
\mathsf{clpo}_6(\mathsf{V}(x), \mathsf{F}(g, ts)) & = & \mathbf{if}\ \mathsf{contains}_{\mathsf{tl}}(ts, x)\ \mathbf{then}\ \mathsf{L}\ \mathbf{else}\ \mathsf{N} \\
\mathsf{clpo}_6(\mathsf{V}(x), \mathsf{V}(y)) & = & \mathbf{if}\ x = y\ \mathbf{then}\ \mathsf{E}\ \mathbf{else}\ \mathsf{N} \\
\mathsf{clpo}_6(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \mathbf{if}\ \ f = g\ \ \mathbf{then}\ \mathsf{cLMA}_6(\mathsf{F}(f, ss), \mathsf{F}(g, ts), ss, ts) \\
 & & \mathbf{elif}\ f >_{\mathsf{F}} g\ \mathbf{then}\ \mathsf{cMA}_6(\mathsf{F}(f, ss), ts) \\
 & & \mathbf{elif}\ g >_{\mathsf{F}} f\ \mathbf{then}\ \mathsf{flip}(\mathsf{cMA}_6(\mathsf{F}(g, ts), ss)) \\
 & & \mathbf{else}\ \mathsf{cAA}_6(\mathsf{F}(f, ss), \mathsf{F}(g, ts), ss, ts)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{cMA}_6 & : & \mathsf{Term}\ \mathsf{Termlist} \to \mathsf{Res} \\
\mathsf{cMA}_6(s, [\,]) & = & \mathsf{G} \\
\mathsf{cMA}_6(s, t \,.\, ts) & = & \mathbf{case}\ \mathsf{clpo}_6(s, t) \\
 & & \quad \mathsf{G}\ \ \ : \mathsf{cMA}_6(s, ts) \\
 & & \quad \mathsf{E}, \mathsf{L} : \mathsf{L} \\
 & & \quad \mathsf{N}\ \ \ : \mathsf{flip}(\mathsf{alpha}_{\mathsf{R6}}(ts, s))
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{cLMA}_6 & : & \mathsf{Term}\ \mathsf{Term}\ \mathsf{Termlist}\ \mathsf{Termlist} \to \mathsf{Res} \\
\mathsf{cLMA}_6(s, t, [\,], [\,]) & = & \mathsf{E} \\
\mathsf{cLMA}_6(s, t, s_i \,.\, ss, t_i \,.\, ts) & = & \mathbf{case}\ \mathsf{clpo}_6(s_i, t_i) \\
 & & \quad \mathsf{E} : \mathsf{cLMA}_6(s, t, ss, ts) \\
 & & \quad \mathsf{G} : \mathsf{cMA}_6(s, ts) \\
 & & \quad \mathsf{L} : \mathsf{flip}(\mathsf{cMA}_6(t, ss)) \\
 & & \quad \mathsf{N} : \mathsf{cAA}_6(s, t, ss, ts)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{cAA}_6 & : & \mathsf{Term}\ \mathsf{Term}\ \mathsf{Termlist}\ \mathsf{Termlist} \to \mathsf{Res} \\
\mathsf{cAA}_6(s, t, ss, ts) & = & \mathbf{case}\ \mathsf{alpha}_{\mathsf{R6}}(ss, t) \\
 & & \quad \mathsf{G} : \mathsf{G} \\
 & & \quad \mathsf{N} : \mathsf{flip}(\mathsf{alpha}_{\mathsf{R6}}(ts, s))
\end{array}
$$

Similar to the previous section, the transformations do not rely on domain specific knowledge about LPO. Whereas they lead to practical improvements of the running time, they do not change the asymptotic behavior. This can be shown with an analysis similar to the analysis of lpo_5.

THEOREM 4.7 *If* $s, t \in \mathsf{Term}$ *are well-formed then* $\mathsf{clpo}_4(s, t) = \mathsf{clpo}_6(s, t)$*. The worst-case running time of* $\mathsf{clpo}_6(s, t)$ *is* $\Theta(|s| \cdot |t|)$*.* □

Further measurements and further variants

The optimizations to achieve a polynomial version of LPO lead to a dramatic improvement in the time needed by LPO. As usual, we use LPO_i to refer to the C-implementation of lpo_i. Table 4.2 shows that LPO_4 is significantly faster than LPO_3 Depending on the example the speedup is between 5 and more than 1000. When we compare the time needed by LPO_4 with the time needed by other operations of the system we see that the overall running time of the prover is no longer dominated by the time needed by the ordering.

Table 4.2: Time (in seconds) needed for ordering comparisons

problem	time needed by ordering							
	LPO$_4$	LPO$_5$	LPO$_6$	LPO$_7$	LPO$_8$	LPO$_9$	LPO$_{10}$	LPO$_{11}$
GRP180-1	0.129	0.124	0.104	0.217	0.177	0.129	0.138	0.242
LAT020-1	0.699	0.762	0.684	1.100	1.099	0.764	0.755	1.115
LCL109-6	0.147	0.149	0.114	0.277	0.213	0.157	0.153	0.273
RNG027-5	0.406	0.419	0.415	0.686	0.570	0.425	0.425	0.733
z22	0.004	0.003	0.003	0.005	0.005	0.003	0.004	0.010
TPTP$_{10}$	81.000	78.700	71.030	126.810	113.090	83.140	85.030	122.410

To our surprise, the transformations leading to lpo_5 show only a small effect on the time requirements. Despite avoiding the separate traversal of syntactic identity tests, LPO$_5$ is not noticeably faster than LPO$_4$. A detailed analysis reveals the following. We have to distinguish two cases: If $s \equiv t$, which accounts for 10 to 20 % of the recursive invocations, LPO$_5$ is slower, because the code is more complicated. If $s \not\equiv t$ then the syntactic equality test detects that pretty fast. On average, it compares less than 2 symbols to find $s \not\equiv t$.[6] This means that on average the overhead of a failing call to $s \equiv t$ is not linear in $|s|$ and $|t|$, but constant. So, for the analyzed examples, the possible gain is rather small, which explains why the transformation does not pay off. Of course, there are examples where $\mathsf{lpo}_{\mathsf{R}5}$ is asymptotically faster than lpo_4, such as $f^n(x) \succ^?_{\text{lpo}} f^m(y)$. However, it seems that in practice the simpler (machine) code of LPO$_4$ outweighs the possible benefits of LPO$_5$.

The optimizations for bidirectional comparisons show, however, moderate improvements. When we compare column LPO$_5$ with column LPO$_6$ we observe improvements of up to 30 %. This is quite remarkable, as in out test cases the majority of calls to the ordering module (about 63 % in TPTP$_{10}$) requires the unidirectional comparison lpo_i, for which LPO$_5$ and LPO$_6$ use identical code. However, for the calls to the bidirectional comparison clpo_i, the majority of results is either L or N (about 89 % in TPTP$_{10}$). In these cases, clpo_5 has to perform two traversals whereas clpo_6 often performs only one (calls to cAA_6 occur quite infrequently). Hence, the optimizations pay off.

The worst-case running times of lpo_5 and lpo_6 are $\Theta(|s| \cdot |t|)$. On p. 39 we sketched a dynamic-programming algorithm which has to determine the ordering relationship between each subterm of s with each subterm of t, i. e., exactly $M = |s| \cdot |t|$ pairs. It is therefore interesting to determine the actual number N of pairs that are compared by the optimized top-down variant LPO$_6$. Figure 4.5 contains the quotient N/M for each invocation of the ordering of example RNG027-5. Only for a small number of invocations is the quotient higher than 0.4, the larger the input, the smaller the quotient. It is therefore highly unlikely that an implementation of the dynamic programming algorithm can be competitive with LPO$_6$.

[6] For most examples the number of symbol comparisons shows a negative-exponential distribution.

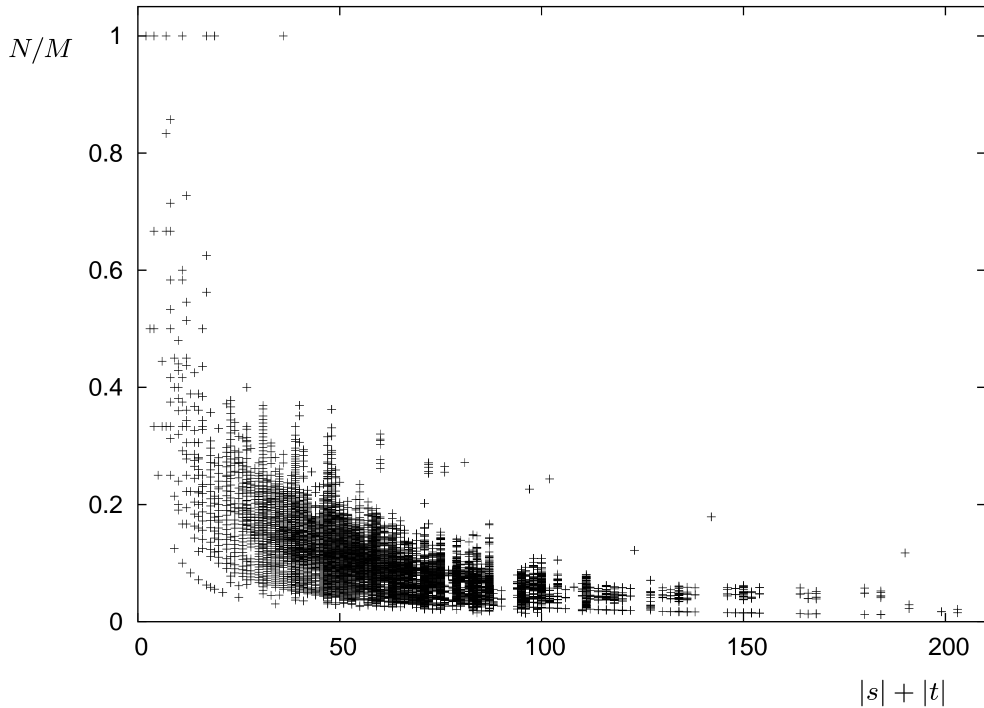


Figure 4.5: Let N be the number of compared pairs of subterms of s and t performed by LPO$_6$ and let $M = |s| \cdot |t|$, the number of compared pairs a dynamic programming version would need. Depicted is N/M over the combined length of s and t for RNG027-5.

Some further variations are worth discussing. We use lpo_4 as a starting point, because it is simpler in structure than lpo_5 and lpo_6. Two variations exploit the following property of LPO: If $s \succ_{\mathrm{lpo}} t$ then $\mathrm{Var}(s) \supseteq \mathrm{Var}(t)$. The idea is to use the contraposition – $\mathrm{Var}(s) \not\supseteq \mathrm{Var}(t)$ implies $s \not\succ_{\mathrm{lpo}} t$ – as a sufficient pretest to restrict or to stop the computation. Bit-oriented machine-instructions allow us to handle a fixed subset $\mathcal{V}_0 \subset \mathcal{V}$ very efficiently. With $|\mathcal{V}_0| = 32$ it fits well the characteristics of the machines we work on (Pentium III). The time to compute $\mathrm{Var}_0(s) = \mathrm{Var}(s) \cap \mathcal{V}_0$ is $\Theta(|s|)$; the time to compare two such variable sets is $O(1)$. Limiting the pretest to $\mathrm{Var}_0(s) \not\supseteq \mathrm{Var}_0(t)$ reduces its power – there might be a variable $x \in \mathrm{Var}(t)$ with $x \notin \mathrm{Var}(s)$ that remains undetected because of $x \notin \mathcal{V}_0$. However, for various reasons WALDMEISTER minimizes the number of different variables in the proof state, hence terms with variables not in $\mathcal{V}_0$ rarely occur in our test cases; the limitation to $\mathcal{V}_0$ should not bias the experiments.

The variant lpo_7. In a preprocessing step we decorate with a recursive procedure each term (including all subterms) with the accompanying variable sets. This is linear in the size of s and t. After that we use in each recursive call the comparison of the variable sets as a pretest. For variables in $\mathcal{V}_0$ the case (δ) becomes trivial to decide and needs $O(1)$ time, which is an additional motivation for this variant. Profiles indicate that using the pretests reduces the running time of the ordering comparisons, but the overhead for the precomputation is not compensated, as the overall times show.

The variant lpo_8. This variant uses variable sets in a weaker form. It is suggested in [Wei01]. Only at the top-level the sets $\mathrm{Var}(s)$ and $\mathrm{Var}(t)$ are determined and used as a pretest for lpo_8 or as a restriction for clpo_8. An analysis shows that the preprocessing

step is now cheaper. For computing the variable sets, we can now use a simple iterative procedure instead of a recursive one. In addition, the write-operations to decorate each subterm with the accompanying variable set are avoided. But the benefits are smaller as well, as can be seen in Table 4.2. In our measurements, the use of variable sets does not pay off.

The following variants try to use additional knowledge that is available in special cases.

The variant lpo_9. This variant uses the so-called "Pacman-Lemma":[7] For unary function symbols f we have $f(s) \succ_{\text{lpo}} f(t)$ iff $s \succ_{\text{lpo}} t$. Then it is possible in lpo to skip the call to lexMA and perform a direct tail-call to itself. We can implement this in an imperative setting by two assignments and a `goto`. Our measurements show that the benefits are tiny, mostly encoded string-rewriting systems profit a little bit, other examples may suffer from a slight slow-down.

The variant lpo_{10}. This variant has specialized code for cases where at least one argument is a constant. Then the rather complex recursion can be replaced by simpler loops. Nevertheless, it does not seem to pay off.

The variant lpo_{11}. Because LPO contains the subterm relation, this variant uses a check for $s \succ_{\text{st}} t$ as a sufficient pretest for $s \succ_{\text{lpo}} t$. This can be a win, because $s \succ_{\text{st}} t$ is cheaper to evaluate. However, our measurements show that the subterm-pretest applies too rarely in practice to justify the extra costs.

When we summarize the measurements we see that lpo_4 shows the best combination of efficiency and coding complexity. Only lpo_6 gives noticeable running-time improvements, but these are moderate (14 % for TPTP_{10}) and the implementation has roughly twice the size. Of the other variants, none shows marked improvements, most are actually disimprovements. But why does folklore suggest them? We guess they are suggested because they help to improve nonoptimal versions. For instance, the "Pacman-Lemma" applied to lpo_3 for some examples gives dramatic improvements over lpo_3. It captures some of the insights that lead from lpo_3 to lpo_4 by avoiding in special situations the call to alpha and thus recomputations that lead to exponential behavior.[8] Nevertheless, the improvements are far from uniform. Applied to the polynomial lpo_4 the potential of these optimizations is much smaller and the overhead of the more complicated code may dominate.

There is one final topic we want to discuss briefly. Up to now we have made no assumption about the term data structure. We conjecture that underlying term representations that keep subterms individual, such as tree-terms, flat-terms, or Prolog-terms (see e. g. [RSV01]), behave quite similarly for LPO comparisons. Our algorithms require access to the top-symbol and to the subterms from left to right, which is sup-

---

[7] This colloquial term stems from the visualization of what happens to common prefixes of the terms under consideration. Take as examples $f(g(f(g(s)))) \succ^?_{\text{lpo}} f(g(f(g(t))))$ – the prefixes are efficiently nibbled away and $s \succ^?_{\text{lpo}} t$ remains.

[8] Consider a variant of the example used in the proof of Theorem 4.4: Let $a >_{\mathcal{F}} b >_{\mathcal{F}} f$ and test $f^n(b) \succ^?_{\text{lpo}} f^n(a)$. The function lpo_3 needs exponential time – the analysis in the proof of Theorem 4.4 applies with small modifications. With the "Pacman-Lemma" this test becomes linear. However, the "Pacman-Lemma" is not applicable for the example that we used in the proof of Theorem 4.4.

ported by all three term representations at similar costs. However, if we have a data structure that supports sharing of identical subterms (i. e., represents the terms in one directed acyclic graph), we can take profit. First, the test for term equality is now a simple pointer comparison. This is especially profitable for variant lpo_4 which then has the same asymptotic complexity as $\mathsf{lpo}_{\mathsf{R5}}$. Second, it is then rather cheap to augment the terms with additional information. So we expect that lpo_7 should be an improvement on lpo_4 in such an environment, as the costs for the explicit computation of the variable sets disappear. Such sets are computed once at term-construction time, the costs are then amortized over many ordering comparisons. Third, it may even become feasible to keep a global history in form of a memo-table of the results of LPO. Then it is possible to share results *between* different invocations of the ordering. But note that here a trade-off applies as well: If the size of the memo-table is to large, the retrieval time increases and may at some point dominate the whole computation.

4.3 The Knuth-Bendix Ordering

Finding an efficient implementation of KBO is not as crucial as for the LPO because the straightforward variant needs quadratic (and not exponential) time in the worst case. Nevertheless, the time spent in KBO can be significant. Schulz gives an estimation of up to 35 % of the overall time for the prover E (personal communication). In [RV04], Riazanov and Voronkov give a figure of about 40 % on the average for the prover Vampire and their straightforward implementation of KBO with up to 80 % for the hardest problems. For Waldmeister we can observe a more modest figure of typically 5–10 % as rewriting with unorientable equations is restricted.

It is widely believed that an implementation of KBO is asymptotically optimal if it shows quadratic worst-case behavior. In the following, however, we will show the derivation of a variant that needs only linear time.

The optimizations we use are solely based on introducing some adequate data structures and modifying the definitions of the functions with some Unfold/Fold-calculus. In contrast to the derivation of an efficient implementation of LPO there are no steps using ordering specific knowledge. To keep the presentation concise (and interesting to read) we omit intermediate steps that occur in the proper use of a program transformation calculus. Instead, for each function we present its initial specification and its final version.

The definition of the KBO

The KBO is named after its first use by Knuth and Bendix in [KB70]. Since then its definition has been slightly generalized by relaxing the variable condition. The KBO is parameterized by a precedence $>_\mathcal{F}$ and a *weight function* $\varphi : \mathcal{F} \cup \mathcal{V} \rightarrow \mathbb{N}$. A weight function φ is *admissible* to a precedence $>_\mathcal{F}$ if there is some $\mu > 0$ such that $\varphi(x) = \mu$ for all $x \in \mathcal{V}$, $\varphi(c) \geq \mu$ for all constants $c \in \mathcal{F}$, and if $f \in \mathcal{F}$ is a unary function symbol with $\varphi(f) = 0$ then $f >_\mathcal{F} g$ for all $g \in \mathcal{F} - \{f\}$. The weight function φ is extended to terms by $\varphi(f(t_1, \ldots, t_n)) = \varphi(f) + \varphi(t_1) + \cdots + \varphi(t_n)$. We assume that there is

at least one constant $c \in \mathcal{F}$ with $\varphi(c) = \mu$ (i. e., μ denotes the weight of the smallest ground term). Recall that a precedence is a partial ordering on $\mathcal{F}$ and that $\mathcal{F}$ contains function symbols of fixed arity only.

Definition 4.2 *Let $>_{\mathcal{F}}$ be a precedence on $\mathcal{F}$, φ a weight function admissible to $>_{\mathcal{F}}$, and $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. Then $s \succ_{\mathrm{kbo}} t$ iff*

- *$s \equiv f(s_1, \ldots, s_n)$, $t \equiv g(t_1, \ldots, t_m)$, and*

(1)	$\lvert s\rvert_x \geq \lvert t\rvert_x$	*for all $x \in \mathcal{V}$ and*
$(2a)$	$\varphi(s) > \varphi(t)$	*or*
$(2b)$	$\varphi(s) = \varphi(t)$, $f >_{\mathcal{F}} g$	*or*
$(2c)$	$\varphi(s) = \varphi(t)$, $f = g$,	*and there is some k with*
	$s_1 \equiv t_1, \ \ldots, \ s_{k-1} \equiv t_{k-1}, \ s_k \succ_{\mathrm{kbo}} t_k$,	

- *or $s \equiv f(s_1, \ldots, s_n)$, $t \equiv x \in \mathcal{V}$, and $x \in \mathrm{Var}(s)$.*

The main idea of the KBO is that each term has a weight and heavier terms are greater than lighter terms; weight function φ allows to differentiate between the different symbols (case $(2a)$).[9] This weight-based ordering is then refined by the precedence (case $(2b)$) and a recursive comparison (case $(2c)$). The variable condition (1) makes $\succ_{\mathrm{kbo}}$ stable against substitutions. All in all, we get the following well-known result, which makes KBO well suited for many applications in term rewriting or saturation-based theorem proving.

Proposition 4.2 *The Knuth-Bendix ordering $\succ_{\mathrm{kbo}}$ with respect to precedence $>_{\mathcal{F}}$ and weight function φ admissible to $>_{\mathcal{F}}$ is a reduction ordering. If $>_{\mathcal{F}}$ is total on $\mathcal{F}$ then $\succ_{\mathrm{kbo}}$ is total on ground terms.* □

By definition 4.2, there are three sub-problems to solve for implementing KBO: the variable condition (1), the computation of the weight $\varphi(s)$, and the lexicographic comparison. Of these, the variable condition is the most complex, whereas implementing $\varphi(s)$ is straightforward:

$$\begin{array}{rcl} \mathsf{phi} & : & \mathsf{Term} \to \mathsf{Nat} \\ \mathsf{phi}(\mathsf{V}(x)) & = & \mu \\ \mathsf{phi}(\mathsf{F}(f, ss)) & = & \varphi(f) + \mathsf{phi}_{\mathsf{tl}}(ss) \end{array} \qquad \begin{array}{rcl} \mathsf{phi}_{\mathsf{tl}} & : & \mathsf{Termlist} \to \mathsf{Nat} \\ \mathsf{phi}_{\mathsf{tl}}([\,]) & = & 0 \\ \mathsf{phi}_{\mathsf{tl}}(s \mathbin{.} ss) & = & \mathsf{phi}(s) + \mathsf{phi}_{\mathsf{tl}}(ss) \end{array}$$

The variable condition (i. e., condition (1) of Definition 4.2) is not effective in that the set $\mathcal{V}$ is infinite by definition. Therefore, several authors restrict the test to $\mathrm{Var}(s,t)$. As this complicates later optimizations, we use a different approach. Whereas $\mathcal{V}$ is infinite, for any invocation of the ordering there exists only a finite subset $\mathcal{V}_{\mathrm{fin}}$ in the state of the prover. Let $K = \lvert\mathcal{V}_{\mathrm{fin}}\rvert$. We assume that the module implementing the KBO can access K and that there is an injective function $\mathsf{index} : \mathsf{Vid} \to \mathsf{Nat}$

[9] Note that φ offers most of the flexibility of KBO: with settings like $\varphi(f) = 0$ or $\varphi(g) = 1\,000\,000$ some extraordinary effects can be achieved.

that transforms efficiently (i. e., in constant time) a variable identifier into a natural number between 1 and K. Then the following function $\mathsf{xlength}$ computes the x-length of a term: $\mathsf{xlength}(s, i) = |s|_x$ iff $\mathsf{index}(x) = i$.

$$
\begin{aligned}
\mathsf{xlength} &: \mathsf{Term\ Nat} \to \mathsf{Nat} \\
\mathsf{xlength}(\mathsf{V}(x), i) &= \textbf{if } \mathsf{index}(x) = i \textbf{ then } 1 \textbf{ else } 0 \\
\mathsf{xlength}(\mathsf{F}(f, ss), i) &= \mathsf{xlength}_{\mathsf{tl}}(ss, i)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{xlength}_{\mathsf{tl}} &: \mathsf{Termlist\ Nat} \to \mathsf{Nat} \\
\mathsf{xlength}_{\mathsf{tl}}([\,], i) &= 0 \\
\mathsf{xlength}_{\mathsf{tl}}(s \centerdot ss, i) &= \mathsf{xlength}(s, i) + \mathsf{xlength}_{\mathsf{tl}}(ss, i)
\end{aligned}
$$

This function allows us to implement function $\mathsf{varCheck}_1$ which checks the variable condition:

$$
\begin{aligned}
\mathsf{varCheck}_1 &: \mathsf{Term\ Term} \to \mathsf{Bool} \\
\mathsf{varCheck}_1(s, t) &= \mathsf{varCheck}'(s, t, 1)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{varCheck}' &: \mathsf{Term\ Term\ Nat} \to \mathsf{Bool} \\
\mathsf{varCheck}'(s, t, i) &= \textbf{if } \ i > K \quad \textbf{then } \mathsf{true} \\
& \ \textbf{elif } \ \mathsf{xlength}(s, i) \ngeq \mathsf{xlength}(t, i) \ \textbf{then } \mathsf{false} \\
& \ \textbf{else } \mathsf{varCheck}'(s, t, i + 1)
\end{aligned}
$$

As $\mathsf{xlength}(s, i)$ needs $O(|s|)$ steps, $\mathsf{varCheck}_1(s, t)$ needs $O(K \cdot (|s| + |t|))$ steps.

With the auxiliary functions available the implementation of KBO is straightforward:

$$
\begin{aligned}
\mathsf{kbo}_1 &: \mathsf{Term\ Term} \to \mathsf{Bool} \\
\mathsf{kbo}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) &= \mathsf{varCheck}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \wedge \\
& \ (\mathsf{phi}(\mathsf{F}(f, ss)) > \mathsf{phi}(\mathsf{F}(g, ts)) \ \vee \\
& \ \ \mathsf{phi}(\mathsf{F}(f, ss)) = \mathsf{phi}(\mathsf{F}(g, ts)) \wedge f >_{\mathsf{F}} g \ \vee \\
& \ \ \mathsf{phi}(\mathsf{F}(f, ss)) = \mathsf{phi}(\mathsf{F}(g, ts)) \wedge f = g \wedge \mathsf{kbolex}_1(ss, ts)) \\
\mathsf{kbo}_1(\mathsf{F}(f, ss), \mathsf{V}(y)) &= \mathsf{contains}_{\mathsf{tl}}(ss, y) \\
\mathsf{kbo}_1(\mathsf{V}(x), t) &= \mathsf{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{kbolex}_1 &: \mathsf{Termlist\ Termlist} \to \mathsf{Bool} \\
\mathsf{kbolex}_1([\,], [\,]) &= \mathsf{false} \\
\mathsf{kbolex}_1(s \centerdot ss, t \centerdot ts) &= \textbf{if } s =_{\mathsf{t}} t \textbf{ then } \mathsf{kbolex}_1(ss, ts) \textbf{ else } \mathsf{kbo}_1(s, t)
\end{aligned}
$$

Note that kbolex_1 is a partial function, it is only defined if the arguments have the same length. This reflects that $\mathcal{F}$ contains function symbols of fixed arity only. As for LPO, we also consider a bidirectional version. Its initial specification is:

$$
\begin{aligned}
\mathsf{ckbo}_1 &: \mathsf{Term\ Term} \to \mathsf{Res} \\
\mathsf{ckbo}_1(s, t) &= \textbf{if } \ s =_{\mathsf{t}} t \quad \textbf{then } \mathsf{E} \\
& \ \textbf{elif } \ \mathsf{kbo}_1(s, t) \textbf{ then } \mathsf{G} \\
& \ \textbf{elif } \ \mathsf{kbo}_1(t, s) \textbf{ then } \mathsf{L} \\
& \ \textbf{else } \mathsf{N}
\end{aligned}
$$

When we analyze the worst-case running time of kbo_1, we see that it is quadratic in the size of the arguments:

Theorem 4.8 *Let s and t be well-formed* Term*s. Then* $\mathsf{kbo}_1(s,t) = \mathsf{true}$ *iff* $s \succ_{\mathrm{kbo}} t$. *The worst-case running time of* $\mathsf{kbo}_1(s,t)$ *is* $O(KN^2)$ *where* $K = |\mathcal{V}_{\mathrm{fin}}|$ *and* $N = |s| + |t|$.

Proof The correctness of the implementation is easy to see as it closely follows Definition 4.2. It remains to consider the worst-case running time. If s is a variable then kbo_1 performs one step. If t is a variable then the function performs $O(|s|)$ steps. In the remaining case, performing the variable check needs $O(KN)$ steps and determining the weights of the terms needs $O(N)$ steps. Furthermore, after testing some subterms for syntactic equality, which is bound linearly in N, the evaluation of kbolex_1 may perform one recursive call to kbo_1. In the worst case, the top-symbol of both terms is a unary function symbol. Hence, for the recursive call the size of each argument is only reduced by one. Therefore, we can estimate the costs of $\mathsf{kbo}_1(s,t)$ by the recurrence $C(N) = c_1KN + c_2N + c_3 + C(N-2)$, where c_1, c_2 and c_3 are some constants. Solving this recurrence leads to $C(N) \leq c'_1KN^2 + c'_2N^2 + c'_3N + c'_4$ for some suitable constants c'_1, c'_2, c'_3, and c'_4. This implies that the worst-case running time of $\mathsf{kbo}_1(s,t)$ is $O(KN^2)$. □

By this analysis, it is easy to come up with instances that show the quadratic behavior of kbo_1. As the number of recursive calls is bound by the depth of the terms, the worst case occurs for terms consisting mostly of unary function symbols. We use KBO_1 to refer to the C-implementation of kbo_1.

Example 4.1 *Let $f >_{\mathcal{F}} a >_{\mathcal{F}} b$ and φ returns 1 for each symbol. Let $u_n \equiv f^n(a)$ and $v_n \equiv f^n(b)$. Then* $\mathsf{kbo}_1(u_n, v_n)$ *needs time that grows linearly in K and quadratically in n to determine $u_n \succ_{\mathrm{kbo}} v_n$. Figure 4.6 depicts the running times of* $\mathrm{KBO}_1(u_n, v_n)$ *for* $n = 1, \dots, 200$ *and six different values of K.*

Of course, the definition of $\mathsf{kbo}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts))$ can easily be optimized by extracting common subexpressions into let-variables and re-bracketing the main expression:

$$
\begin{aligned}
&\mathsf{kbo}(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \ = \\
&\quad \textbf{let}\ \ vc = \mathsf{varCheck}_1(\mathsf{F}(f, ss), \mathsf{F}(g, ts)) \\
&\quad\quad\quad w_s = \mathsf{phi}(\mathsf{F}(f, ss)) \\
&\quad\quad\quad w_t = \mathsf{phi}(\mathsf{F}(g, ts)) \\
&\quad \textbf{in}\ \ vc \wedge (w_s > w_t \vee (w_s = w_t \wedge (f >_{\mathsf{F}} g \vee (f = g \wedge \mathsf{kbolex}(ss, ts)))))
\end{aligned}
$$

However, this modification only improves the constants, the asymptotic running time is still $O(KN^2)$ in the worst case. The main problems are the expensive check of the variable condition and the repeated work induced by recursive calls.

As explained on p. 39, standard techniques to avoid repeated computations such as dynamic programming or memoization use an additional data structure to store intermediate results. This is unproblematic for the weights of the subterms. However, to

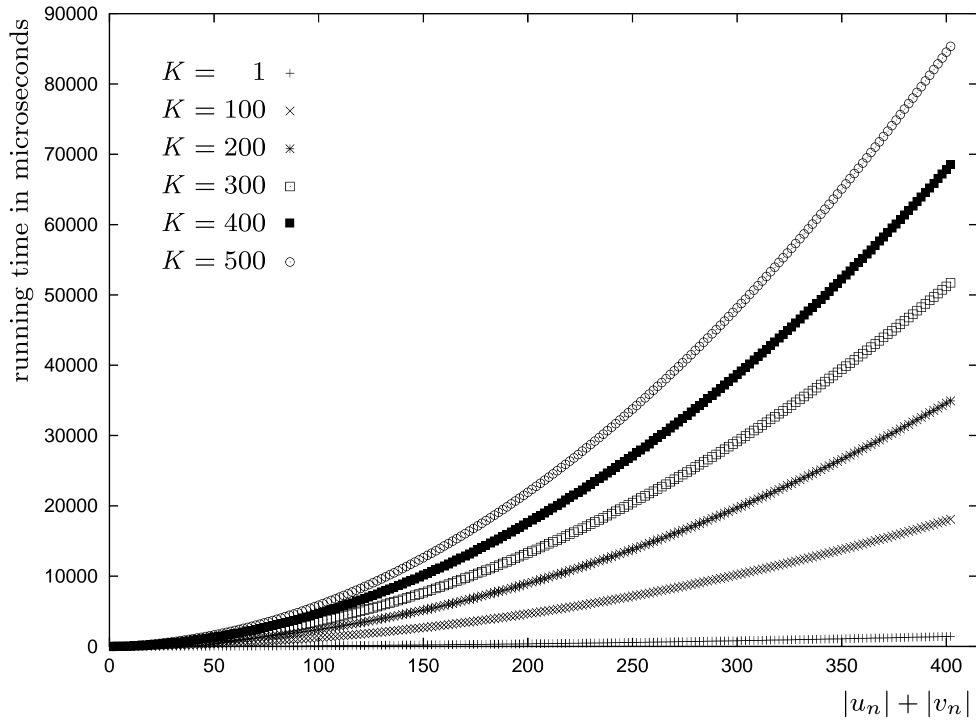


Figure 4.6: Time needed to evaluate $\text{KBO}_1(u_n, v_n)$ with $n = 1, \ldots, 200$ and six different values of K (cf. Example 4.1)

implement the variable condition, we have to store the multisets of variables occurring in a subterm. The following example shows that this may lead to space problems.

EXAMPLE 4.2 *Let $g >_{\mathcal{F}} a >_{\mathcal{F}} b$ and φ return 1 for each symbol. The terms u_n are given by $u_0 \equiv a$ and $u_{n+1} \equiv g(u_n, x_n, y_n)$. The terms v_n are given by $v_0 \equiv b$ and $v_{n+1} \equiv g(v_n, y_n, x_n)$. Then $u_n \succ_{\text{kbo}} v_n$. However, the multisets of variables for u_n and v_n contain $2n$ different variables. This means that simply storing the multisets of variables for each term leads to a quadratic memory requirement.*

It is easy to develop representations of the multisets that circumvent the quadratic memory requirement for this example. To find general solutions, however, is quite challenging. As it turns out, we can derive an efficient version of KBO without the use of a memo-table. First, we want to improve the check of the variable condition.

Implementing the variable condition with arrays

The main problem with function $\mathsf{varCheck}_1$ is that it needs K traversals of each term. An obvious way to avoid this is to compute the multisets of variables contained in s and t and to test whether the multiset belonging to t is a sub-multiset of the multiset belonging to s. This needs only one traversal per term. However, the use of multisets complicates later transformations. Hence, we use a different approach which is based on arrays and which is also closer to an imperative implementation. The arrays contain

one entry for each variable. By using function index we can efficiently access the entry for a given variable.

In the imperative implementation arrays are modified destructively. To model this, we thread arrays as additional parameters through the function calls and require that a variable bound to an array must be used exactly once in each branch of the function definition.[10] This facilitates a linearity analysis that some compilers of functional languages use as an optimization. Instead of being copied, linearly used data structures can be modified destructively.

It is therefore convenient to reduce the number of arrays. Instead of two arrays of naturals for the number of variables in s and the number of variables in t we can use one single array of integers which stores for each variable x the *variable balance* $|s|_x - |t|_x$. The test of condition (1) of Definition 4.2 is then replaced by $|s|_x - |t|_x \geq 0$ for all $x \in \mathcal{V}_{\text{fin}}$.

To enhance modularity and ease later optimizations we use a dedicated data type VarBal for variable balances with corresponding methods. We use **1** to denote the type of the empty tuple with the single value $\langle\rangle$, which plays the same role as `void` in the C language. The following two functions allocate and deallocate an instance of VarBal. We assume that $\mathsf{newArray}(\mathsf{Int}, 1, K)$ returns a new array of integers with indices ranging from 1 to K such that all entries are initialized to 0. We write $vb = \vec{0}$ to denote that all entries of vb have value 0. Function $\mathsf{freeArray}$ frees in some unspecified way the memory occupied by the array passed as parameter.

$$\begin{array}{rcl} \mathsf{newVB} & : & \mathbf{1} \rightarrow \mathsf{VarBal} \\ \mathsf{newVB}(\langle\rangle) & = & \mathsf{newArray}(\mathsf{Int}, 1, K) \end{array} \qquad \begin{array}{rcl} \mathsf{freeVB} & : & \mathsf{VarBal} \rightarrow \mathbf{1} \\ \mathsf{freeVB}(vb) & = & \mathsf{freeArray}(vb) \end{array}$$

A conservative estimation is that both functions run in $O(K)$ time. We will discuss later about how to improve on this (see pp. 68–71).

To increment and decrement the entry for a given variable in a VarBal we use the following two functions:

$$\begin{array}{rcl} \mathsf{inc} & : & \mathsf{VarBal}\ \mathsf{Vid} \rightarrow \mathsf{VarBal} \\ \mathsf{inc}(vb, x) & = & \mathbf{let}\ \ i = \mathsf{index}(x) \\ & & \quad \langle vb', n\rangle = \mathsf{read}(vb, i) \\ & & \mathbf{in}\ \ \mathsf{update}(vb', i, n+1) \end{array} \qquad \begin{array}{rcl} \mathsf{dec} & : & \mathsf{VarBal}\ \mathsf{Vid} \rightarrow \mathsf{VarBal} \\ \mathsf{dec}(vb, x) & = & \mathbf{let}\ \ i = \mathsf{index}(x) \\ & & \quad \langle vb', n\rangle = \mathsf{read}(vb, i) \\ & & \mathbf{in}\ \ \mathsf{update}(vb', i, n-1) \end{array}$$

Note the explicit threading of the array. Functions read and update are standard functions to read and modify entries of an array. We assume that they need constant time. Therefore, functions inc and dec need constant time as well.

The following function noNeg tests whether for all variables the balance is not negative (i. e., $|s|_x - |t|_x \geq 0$ for all $x \in \mathcal{V}_{\text{fin}}$). Conversely, function noPos tests whether for all variables the balance is not positive. This is useful for the bidirectional version of KBO.

[10] The cognoscenti will recognize a state monad [Wad92].

$$\begin{array}{rcl} \mathsf{noNeg} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noNeg}(vb) & = & \mathsf{noNeg}'(vb, 1) \end{array} \qquad \begin{array}{rcl} \mathsf{noPos} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noPos}(vb) & = & \mathsf{noPos}'(vb, 1) \end{array}$$

$$\begin{array}{rcl} \mathsf{noNeg}' & : & \mathsf{VarBal}\ \mathsf{Nat} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noNeg}'(vb, i) & = & \\ \multicolumn{3}{l}{\quad \textbf{if } i > K \textbf{ then } \langle vb, \mathsf{true} \rangle} \\ \multicolumn{3}{l}{\quad \textbf{else let } \langle vb', n \rangle = \mathsf{read}(vb, i)} \\ \multicolumn{3}{l}{\qquad \textbf{in } \textbf{if } n < 0 \textbf{ then } \langle vb', \mathsf{false} \rangle} \\ \multicolumn{3}{l}{\qquad\quad \textbf{else } \mathsf{noNeg}'(vb', i+1)} \end{array} \qquad \begin{array}{rcl} \mathsf{noPos}' & : & \mathsf{VarBal}\ \mathsf{Nat} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noPos}'(vb, i) & = & \\ \multicolumn{3}{l}{\quad \textbf{if } i > K \textbf{ then } \langle vb, \mathsf{true} \rangle} \\ \multicolumn{3}{l}{\quad \textbf{else let } \langle vb', n \rangle = \mathsf{read}(vb, i)} \\ \multicolumn{3}{l}{\qquad \textbf{in } \textbf{if } n > 0 \textbf{ then } \langle vb', \mathsf{false} \rangle} \\ \multicolumn{3}{l}{\qquad\quad \textbf{else } \mathsf{noPos}'(vb', i+1)} \end{array}$$

It is easy to see that the worst-case running time of both functions is $O(K)$. Note that only functions newVB, freeVB, inc, dec, noNeg and noPos rely on the knowledge of how the data type VarBal is represented.

Function calcVB modifies the variable balances in one traversal. The argument *pos* indicates whether calcVB has to increase or to decrease the entries in vb.

$$\begin{array}{rcl} \mathsf{calcVB} & : & \mathsf{VarBal}\ \mathsf{Term}\ \mathsf{Bool} \to \mathsf{VarBal} \\ \mathsf{calcVB}(vb, \mathsf{V}(x), pos) & = & \textbf{if } pos \textbf{ then } \mathsf{inc}(vb, x) \\ & & \textbf{else } \mathsf{dec}(vb, x) \\ \mathsf{calcVB}(vb, \mathsf{F}(f, ss), pos) & = & \mathsf{calcVB}_{\mathsf{tl}}(vb, ss, pos) \\[1ex] \mathsf{calcVB}_{\mathsf{tl}} & : & \mathsf{VarBal}\ \mathsf{Termlist}\ \mathsf{Bool} \to \mathsf{VarBal} \\ \mathsf{calcVB}_{\mathsf{tl}}(vb, [\,], pos) & = & vb \\ \mathsf{calcVB}_{\mathsf{tl}}(vb, s \centerdot ss, pos) & = & \textbf{let } vb' = \mathsf{calcVB}(vb, s, pos) \\ & & \textbf{in } \ \mathsf{calcVB}_{\mathsf{tl}}(vb', ss, pos) \end{array}$$

The running time of functions calcVB and $\mathsf{calcVB}_{\mathsf{tl}}$ is linear in the size of the second argument.

With these functions it is possible to implement function $\mathsf{varCheck}_2$:

$$\begin{array}{rcl} \mathsf{varCheck}_2 & : & \mathsf{Term}\ \mathsf{Term} \to \mathsf{Bool} \\ \mathsf{varCheck}_2(s, t) & = & \textbf{let} \quad \begin{array}{rcl} vb & = & \mathsf{newVB}(\langle\rangle) \\ vb' & = & \mathsf{calcVB}(vb, s, \mathsf{true}) \\ vb'' & = & \mathsf{calcVB}(vb', t, \mathsf{false}) \\ \langle vb''', res \rangle & = & \mathsf{noNeg}(vb'') \\ \langle\rangle & = & \mathsf{freeVB}(vb''') \end{array} \\ & & \textbf{in } \ res \end{array}$$

Taking into account the running times of the functions concerning VarBal it is clear that the worst-case running time of $\mathsf{varCheck}_2(s,t)$ is in $O(K + |s| + |t|)$.

Function kbo_2 (and hence ckbo_2) differs from function kbo_1 in using $\mathsf{varCheck}_2$ instead of $\mathsf{varCheck}_1$ and having the improved definition for $\mathsf{kbo}(\mathsf{F}(f, ss), \mathsf{F}(g, ts))$, thereby avoiding the multiple computations of identical weights.

THEOREM 4.9 *Let s and t be well-formed* Terms*. Then* $\mathsf{kbo}_2(s,t) = \mathsf{kbo}_1(s,t)$*. The worst-case running time of* $\mathsf{kbo}_2(s,t)$ *is* $O(KN + N^2)$ *where* $K = |\mathcal{V}_{\mathrm{fin}}|$ *and* $N = |s| + |t|$.

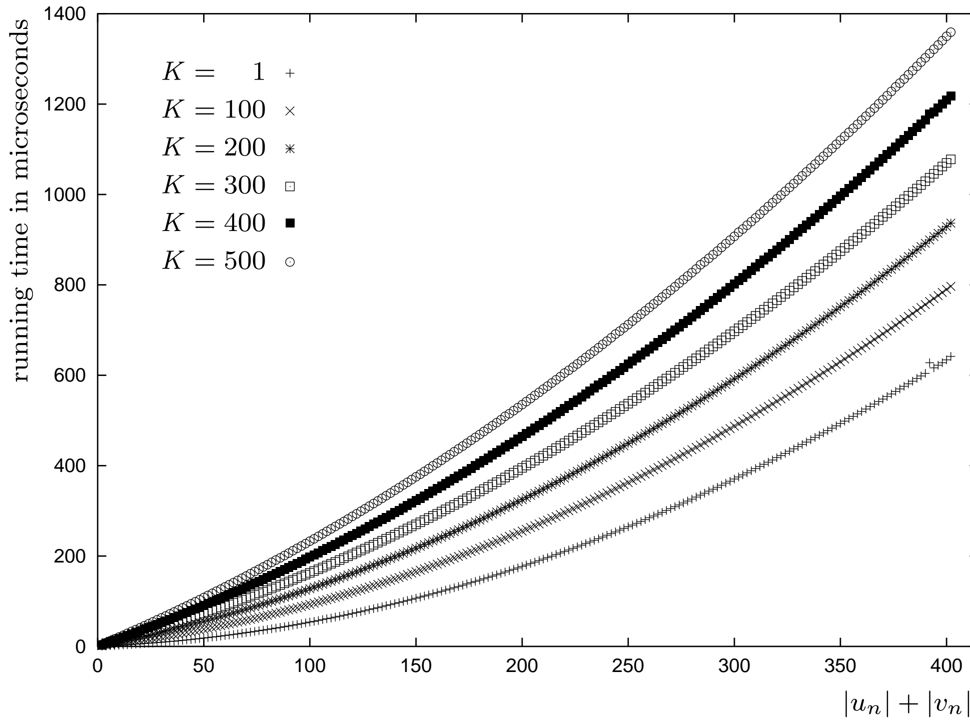


Figure 4.7: Time needed to evaluate $\text{KBO}_2(u_n, v_n)$ with $n = 1, \ldots, 200$ and six different values of K (cf. Example 4.1)

PROOF It is easy to see that $\mathsf{varCheck}_2(s,t) = \mathsf{varCheck}_1(s,t)$ and that the modified definition for $\mathsf{kbo}_2(\mathsf{F}(f,ss),\mathsf{F}(g,ts))$ is equivalent to the original one. It remains to consider the worst-case running time. If at least one of the arguments is a variable then $\mathsf{kbo}_2(s,t)$ behaves identical to $\mathsf{kbo}_1(s,t)$: Its running time is either $O(1)$ or $O(|s|)$. In the remaining case, performing the variable check with $\mathsf{varCheck}_2$ needs $O(K+N)$ steps. Determining the weights of the terms still is $O(N)$. (However, the constants are smaller than for kbo_1 as repeated work is avoided). For the evaluation of kbolex nothing changes. Therefore, we can estimate the costs of $\mathsf{kbo}_2(s,t)$ by the recurrence $C(N) = c_1K + c_2N + c_3 + C(N-2)$, where c_1, c_2 and c_3 are some constants. Solving this recurrence leads to $C(N) \leq c'_1KN + c'_2N^2 + c'_3N + c'_4$ for some suitable constants c'_1, c'_2, c'_3 and c'_4. This implies that the worst-case running time of $\mathsf{kbo}_2(s,t)$ is $O(KN+N^2)$. $\square$

Revisiting Example 4.1, we see in Figure 4.7 that KBO_2, the C-implementation of kbo_2, is significantly faster than KBO_1 for these test cases. (Figure 4.7 has a significantly smaller scale than Figure 4.6). Note the different slope of the graphs depicted in Figures 4.6 and 4.7 which indicates the different dependency on K of the two implementations.

Deriving a linear version

The *tupling strategy* is a standard approach in program transformations [PP93]. The general idea is to combine several independent sub-computations into a new function

returning a tuple that contains the results of these sub-computations. When it is used to avoid multiple traversals of data structures it is usually an instance of the "banana-split" theorem [BdM97]. Changes in the asymptotic complexity are achieved, if the combination of the sub-computations allows us to reuse intermediate results, thus avoiding recomputations.

The key insight in optimizing kbo_2 is that the tupling strategy allows us not only to combine the calculation of the variable balances and the weights, which is straightforward, but also to include the lexicographic comparison. Hence, all three sub-computations are combined into one traversal of the terms. As this avoids the iterated computations of variable balances and weights it leads to an algorithm that is linear in $|s| + |t|$.

To achieve the desired result we need a variant of kbo that incorporates the tests for syntactic equality (similar to $\mathsf{lpo}_{\mathsf{R5}}$). As it turns out, developing a version for the bidirectional ckbo is only slightly more involved than developing a version for a three-valued unidirectional variant. For the presentation, we therefore prefer the bidirectional variant for its greater symmetry. The unidirectional variant can easily be derived by simplifying the bidirectional one.

Because of the variable balances, we have to thread some state through the computation. By the tupling strategy the state is extended to incorporate the weights of the terms and the result of the lexicographic comparison. To reduce the number of components of the state, we therefore combine both weights into one *weight balance* wb which is defined as $wb = \varphi(s) - \varphi(t)$. Analogously to the variable balances, the test $\varphi(s) \geq \varphi(t)$ is then replaced by $wb \geq 0$. To get used to the tupling approach we first define some auxiliary functions. Function calcVWB combines the modification of the variable balances with the modification of the weight balance. Its initial definition is the following:

$$\begin{aligned}
\mathsf{calcVWB}(vb, wb, s, pos) \;=\; & \textbf{let } vb' = \mathsf{calcVB}(vb, s, pos) \\
& \phantom{\textbf{let }} w = \mathsf{phi}(s) \\
& \textbf{in if } pos \textbf{ then } \langle vb', wb + w\rangle \textbf{ else } \langle vb', wb - w\rangle
\end{aligned}$$

With the tupling strategy we can avoid the multiple traversals of the term:

$$\begin{aligned}
\mathsf{calcVWB} &: \mathsf{VarBal\ Int\ Term\ Bool} \to \mathsf{VarBal} \times \mathsf{Int} \\
\mathsf{calcVWB}(vb, wb, \mathsf{V}(x), pos) &= \textbf{if } pos \textbf{ then } \langle \mathsf{inc}(vb, x), wb + \mu\rangle \\
& \textbf{else } \langle \mathsf{dec}(vb, x), wb - \mu\rangle \\
\mathsf{calcVWB}(vb, wb, \mathsf{F}(f, ss), pos) &= \textbf{let } \langle vb', wb'\rangle = \mathsf{calcVWB_{tl}}(vb, wb, ss, pos) \\
& \textbf{in if } pos \textbf{ then } \langle vb', wb' + \varphi(f)\rangle \\
& \qquad \textbf{else } \langle vb', wb' - \varphi(f)\rangle \\
\mathsf{calcVWB_{tl}} &: \mathsf{VarBal\ Int\ Termlist\ Bool} \to \mathsf{VarBal} \times \mathsf{Int} \\
\mathsf{calcVWB_{tl}}(vb, wb, [\,], pos) &= \langle vb, wb\rangle \\
\mathsf{calcVWB_{tl}}(vb, wb, s \,.\, ss, pos) &= \textbf{let } \langle vb', wb'\rangle = \mathsf{calcVWB}(vb, wb, s, pos) \\
& \textbf{in } \mathsf{calcVWB_{tl}}(vb', wb', ss, pos)
\end{aligned}$$

Obviously, evaluating $\mathsf{calcVWB}(vb, wb, s, pos)$ needs $O(|s|)$ steps.

Function $\mathsf{calcVWBc}$ integrates in addition the test whether some variable symbol y is contained in the term s:

$$\begin{array}{rcl}\mathsf{calcVWBc}(vb, wb, s, y, pos) & = & \textbf{let}\ \langle vb', wb'\rangle = \mathsf{calcVWB}(vb, wb, s, pos) \\ & & \qquad\qquad\quad res = \mathsf{contains}(s, y) \\ & & \textbf{in}\ \ \langle vb', wb', res\rangle\end{array}$$

With tupling we get the following variant:

$$\begin{array}{rcl}\mathsf{calcVWBc} & : & \mathsf{VarBal\ Int\ Term\ Vid\ Bool} \rightarrow \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Bool} \\ \mathsf{calcVWBc}(vb, wb, \mathsf{V}(x), y, pos) & = & \textbf{if}\ pos\ \textbf{then}\ \langle \mathsf{inc}(vb, x), wb + \mu, x = y\rangle \\ & & \textbf{else}\ \langle \mathsf{dec}(vb, x), wb - \mu, x = y\rangle \\ \mathsf{calcVWBc}(vb, wb, \mathsf{F}(f, ss), y, pos) & = & \textbf{let}\ \langle vb', wb', res\rangle = \mathsf{calcVWBc_{tl}}(vb, wb, ss, y, pos) \\ & & \textbf{in}\ \ \textbf{if}\ pos\ \textbf{then}\ \langle vb', wb' + \varphi(f), res\rangle \\ & & \qquad \textbf{else}\ \langle vb', wb' - \varphi(f), res\rangle \\ \mathsf{calcVWBc_{tl}} & : & \mathsf{VarBal\ Int\ Termlist\ Vid\ Bool} \ \rightarrow \mathsf{VarBal} \times \mathsf{Int} \\ \mathsf{calcVWBc_{tl}}(vb, wb, [\,], y, pos) & = & \langle vb, wb, \mathsf{false}\rangle \\ \mathsf{calcVWBc_{tl}}(vb, wb, s \centerdot ss, y, pos) & = & \textbf{let}\ \langle vb', wb', res\rangle = \mathsf{calcVWBc}(vb, wb, s, y, pos) \\ & & \textbf{in}\ \ \textbf{if}\ \neg res\ \textbf{then}\ \mathsf{calcVWBc_{tl}}(vb', wb', ss, y, pos) \\ & & \qquad \textbf{else}\ \textbf{let}\ \langle vb'', wb''\rangle = \mathsf{calcVWB_{tl}}(vb', wb', ss, pos) \\ & & \qquad\qquad \textbf{in}\ \ \langle vb'', wb'', \mathsf{true}\rangle\end{array}$$

As each symbol in the term is considered once at constant costs, the evaluation of $\mathsf{calcVWBc}(vb, wb, s, y, pos)$ needs $O(|s|)$ steps.

The tupled variant of ckbo modifies the given variable and weight balances and computes ckbo of the terms:

$$\begin{array}{rcl}\mathsf{tckbo}(vb, wb, s, t) & = & \textbf{let}\ \ \langle vb', wb'\rangle = \mathsf{calcVWB}(vb, wb, s, \mathsf{true}) \\ & & \qquad \langle vb'', wb''\rangle = \mathsf{calcVWB}(vb', wb', t, \mathsf{false}) \\ & & \qquad\qquad\quad res = \mathsf{ckbo}(s, t) \\ & & \textbf{in}\ \ \langle vb'', wb'', res\rangle\end{array}$$

Note that if $s =_{\mathsf{t}} t$ then $vb'' = vb$ and $wb'' = wb$ (i.e., the possible modifications of each variable balance and the weight balance cancel each other).

Similar to tckbo, we need a tupled variant of the lexicographic comparison. We first specify a Res-valued variant of the lexicographic comparison. Like function kbolex it is only defined for $\mathsf{Termlists}$ of the same length.

$$\begin{array}{rcl}\mathsf{ckbolex}(ss, ts) & = & \textbf{if}\quad ss =_{\mathsf{tl}} ts \qquad \textbf{then}\ \mathsf{E} \\ & & \textbf{elif}\ \ \mathsf{kbolex}(ss, ts)\ \textbf{then}\ \mathsf{G} \\ & & \textbf{elif}\ \ \mathsf{kbolex}(ts, ss)\ \textbf{then}\ \mathsf{L} \\ & & \textbf{else}\ \mathsf{N}\end{array}$$

It is easy to see that the following variant is equivalent:

$$\begin{array}{rcl}\mathsf{ckbolex}([\,], [\,]) & = & \mathsf{E} \\ \mathsf{ckbolex}(s \centerdot ss, t \centerdot ts) & = & \textbf{let}\ res = \mathsf{ckbo}(s, t) \\ & & \textbf{in}\ \ \textbf{if}\ res = \mathsf{E}\ \textbf{then}\ \mathsf{ckbolex}(ss, ts) \\ & & \qquad \textbf{else}\ res\end{array}$$

The tupled variant $\mathsf{tckbolex}$ takes into account the variable and weight balances. We have to ensure that it is only called with $\mathsf{Termlists}$ of the same length as otherwise it is undefined.

$$\begin{aligned}
\mathsf{tckbolex}(vb, wb, ss, ts) \;=\; & \textbf{let}\;\; \langle vb', wb'\rangle = \mathsf{calcVWB_{tl}}(vb, wb, ss, \mathsf{true}) \\
& \qquad \langle vb'', wb''\rangle = \mathsf{calcVWB_{tl}}(vb', wb', ts, \mathsf{false}) \\
& \qquad\qquad res = \mathsf{ckbolex}(ss, ts) \\
& \textbf{in}\;\; \langle vb'', wb'', res\rangle
\end{aligned}$$

Note that if $s =_{\mathsf{t}} t$ then $\mathsf{tckbolex}(vb, wb, s \,.\, ss, t \,.\, ts)$ returns the same result as $\mathsf{tckbolex}(vb, wb, ss, ts)$.

In the specification of tckbo and $\mathsf{tckbolex}$ the calculation of the ordering relation and the modification of the variable and weight balances occur independently. However, the whole point of the optimizations is to perform them simultaneously and to use the variable and weight balances for computing the ordering relations. To get correct results we therefore have to establish the following invariant I: At each invocation of tckbo and $\mathsf{tckbolex}$ both functions are called with $vb = \vec{0}$ and $wb = 0$. Thus, it is sufficient that the optimized versions of tckbo and $\mathsf{tckbolex}$ give the same results as the unoptimized versions only if called with $vb = \vec{0}$ and $wb = 0$.

The optimized version of $\mathsf{tckbolex}$ is called $\mathsf{tckbolex}_3$. Note the close relationship to the second variant of $\mathsf{ckbolex}$.

$$\begin{aligned}
\mathsf{tckbolex}_3 \;:\; & \mathsf{VarBal\ Int\ Termlist\ Termlist} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Res} \\
\mathsf{tckbolex}_3(vb, wb, [\,], [\,]) \;=\; & \langle vb, wb, \mathsf{E}\rangle \\
\mathsf{tckbolex}_3(vb, wb, s \,.\, ss, t \,.\, ts) \;=\; & \textbf{let}\; \langle vb', wb', res\rangle = \mathsf{tckbo}_3(vb, wb, s, t) \\
& \textbf{in}\;\; \textbf{if}\; res = \mathsf{E}\; \textbf{then}\; \mathsf{tckbolex}_3(vb', wb', ss, ts) \\
& \qquad \textbf{else let}\;\; \langle vb'', wb''\rangle = \mathsf{calcVWB_{tl}}(vb', wb', ss, \mathsf{true}) \\
& \qquad\qquad\quad \langle vb''', wb'''\rangle = \mathsf{calcVWB_{tl}}(vb'', wb'', ts, \mathsf{false}) \\
& \qquad\qquad \textbf{in}\;\; \langle vb''', wb''', res\rangle
\end{aligned}$$

In the following, we describe the optimized version of tckbo, which admittedly looks rather complex. However, most of the complexity comes from the threading of the state. The complicated-looking **if**-**elif**-expression simply checks the different possibilities in a systematic way.

$$\begin{aligned}
\mathsf{tckbo}_3 \;:\; & \mathsf{VarBal\ Int\ Term\ Term} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Res} \\
\mathsf{tckbo}_3(vb, wb, \mathsf{V}(x), \mathsf{V}(y)) \;=\; & \textbf{let}\;\; vb' = \mathsf{inc}(vb, x) \\
& \qquad vb'' = \mathsf{dec}(vb', y) \\
& \qquad res = \textbf{if}\; x = y\; \textbf{then}\; \mathsf{E}\; \textbf{else}\; \mathsf{N} \\
& \textbf{in}\;\; \langle vb'', wb, res\rangle \\
\mathsf{tckbo}_3(vb, wb, \mathsf{V}(x), \mathsf{F}(g, ts)) \;=\; & \textbf{let}\; \langle vb', wb', ctn\rangle = \mathsf{calcVWBc}(vb, wb, \mathsf{F}(g, ts), x, \mathsf{false}) \\
& \qquad\qquad res = \textbf{if}\; ctn\; \textbf{then}\; \mathsf{L}\; \textbf{else}\; \mathsf{N} \\
& \qquad\qquad vb'' = \mathsf{inc}(vb', x) \\
& \textbf{in}\;\; \langle vb'', wb' + \mu, res\rangle \\
\mathsf{tckbo}_3(vb, wb, \mathsf{F}(f, ss), \mathsf{V}(y)) \;=\; & \textbf{let}\; \langle vb', wb', ctn\rangle = \mathsf{calcVWBc}(vb, wb, \mathsf{F}(f, ss), y, \mathsf{true}) \\
& \qquad\qquad res = \textbf{if}\; ctn\; \textbf{then}\; \mathsf{G}\; \textbf{else}\; \mathsf{N} \\
& \qquad\qquad vb'' = \mathsf{dec}(vb', y) \\
& \textbf{in}\;\; \langle vb'', wb' - \mu, res\rangle
\end{aligned}$$

$$
\begin{array}{rcl}
\mathsf{tckbo}_3(vb, wb, \mathsf{F}(f, ss), \mathsf{F}(g, ts)) & = & \textbf{let } \langle vb', wb', lex\rangle = \mathsf{tckbo}'_3(vb, wb, f, g, ss, ts) \\
& & \qquad wb'' = wb' + \varphi(f) - \varphi(g) \\
& & \qquad \langle vb'', nNeg\rangle = \mathsf{noNeg}(vb') \\
& & \qquad \langle vb''', nPos\rangle = \mathsf{noPos}(vb'') \\
& & \qquad G\text{-}or\text{-}N = \textbf{if } nNeg \textbf{ then } \mathsf{G} \textbf{ else } \mathsf{N} \\
& & \qquad L\text{-}or\text{-}N = \textbf{if } nPos \textbf{ then } \mathsf{L} \textbf{ else } \mathsf{N} \\
& & \textbf{in if } \; wb'' > 0 \textbf{ then } \langle vb''', wb'', G\text{-}or\text{-}N\rangle \\
& & \quad \textbf{elif } wb'' < 0 \textbf{ then } \langle vb''', wb'', L\text{-}or\text{-}N\rangle \\
& & \quad \textbf{elif } f >_{\mathsf{F}} g \textbf{ then } \langle vb''', wb'', G\text{-}or\text{-}N\rangle \\
& & \quad \textbf{elif } g >_{\mathsf{F}} f \textbf{ then } \langle vb''', wb'', L\text{-}or\text{-}N\rangle \\
& & \quad \textbf{elif } f \neq g \textbf{ then } \langle vb''', wb'', \mathsf{N}\rangle \\
& & \quad \textbf{elif } lex = \mathsf{E} \textbf{ then } \langle vb''', wb'', \mathsf{E}\rangle \\
& & \quad \textbf{elif } lex = \mathsf{G} \textbf{ then } \langle vb''', wb'', G\text{-}or\text{-}N\rangle \\
& & \quad \textbf{elif } lex = \mathsf{L} \textbf{ then } \langle vb''', wb'', L\text{-}or\text{-}N\rangle \\
& & \quad \textbf{else } \langle vb''', wb'', \mathsf{N}\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{tckbo}'_3 & : & \mathsf{VarBal\ Int\ Fid\ Fid\ Termlist\ Termlist} \to \mathsf{VarBal} \times \mathsf{Int} \times \mathsf{Res} \\
\mathsf{tckbo}'_3(vb, wb, f, g, ss, ts) & = & \textbf{if } f = g \textbf{ then } \mathsf{tckbolex}_3(vb, wb, ss, ts) \\
& & \textbf{else let } \langle vb', wb'\rangle = \mathsf{calcVWB}_{\mathsf{tl}}(vb, wb, ss, \mathsf{true}) \\
& & \qquad \langle vb'', wb''\rangle = \mathsf{calcVWB}_{\mathsf{tl}}(vb', wb', ts, \mathsf{false}) \\
& & \quad \textbf{in } \langle vb'', wb'', \mathsf{N}\rangle
\end{array}
$$

Note that $\mathsf{tckbolex}_3$ is only called if $f = g$ which implies that ss and ts have the same length. Analogously, the value of lex is only considered if $f = g$ (i. e., results from invoking $\mathsf{tckbolex}_3$).

LEMMA 4.8 *Let* $vb = \vec{0}$ *and* $wb = 0$*. If* $s =_{\mathsf{t}} t$ *then* $\mathsf{tckbo}_3(vb, wb, s, t) = \langle\vec{0}, 0, \mathsf{E}\rangle$*. If* $ss =_{\mathsf{tl}} ts$ *then* $\mathsf{tckbolex}_3(vb, wb, ss, ts) = \langle\vec{0}, 0, \mathsf{E}\rangle$.

PROOF Simultaneous induction on $|s| + |t|$ and $|ss| + |ts|$. If $s =_{\mathsf{t}} t$ then either $s = \mathsf{V}(x)$, $t = \mathsf{V}(y)$, and $x = y$, or $s = \mathsf{F}(f, ss)$, $t = \mathsf{F}(g, ts)$, $f = g$, and $ss =_{\mathsf{tl}} ts$. In the first case, tckbo_3 first increments, then decrements the same entry of vb, then sets res to E and uses wb unmodified. Hence, it returns $\langle\vec{0}, 0, \mathsf{E}\rangle$. In the second case, tckbo'_3 calls $\mathsf{tckbolex}_3$ which returns $\langle\vec{0}, 0, \mathsf{E}\rangle$ by induction hypothesis. By adding and subtracting the same value the weight balance is not changed. Hence, tckbo_3 returns $\langle\vec{0}, 0, \mathsf{E}\rangle$.

If $ss =_{\mathsf{tl}} ts$ then either both lists are $[\,]$ or $ss = s \centerdot ss'$, $ts = t \centerdot ts'$, $s =_{\mathsf{t}} t$, and $ss' =_{\mathsf{tl}} ts'$. In the first case, $\mathsf{tckbolex}_3$ lets vb and wb unmodified. Hence, it returns $\langle\vec{0}, 0, \mathsf{E}\rangle$. In the second case, it calls tckbo_3 which by induction hypothesis returns $\langle\vec{0}, 0, \mathsf{E}\rangle$. Then $\mathsf{tckbolex}_3$ calls itself with $vb' = \vec{0}$ and $wb' = 0$. Hence, induction hypothesis applies, the result is $\langle\vec{0}, 0, \mathsf{E}\rangle$. □

Because recursive calls of tckbo_3 and $\mathsf{tckbolex}_3$ occur only as long as the symbols of s and t are identical, we immediately get the following corollary.

COROLLARY 4.2 *The invariant* I *is established by* tckbo_3 *and* $\mathsf{tckbolex}_3$. □

LEMMA 4.9 *Let* $vb = \vec{0}$*,* $wb = 0$ *and* $\mathsf{length}(ss) = \mathsf{length}(ts)$*. Then* $\mathsf{tckbo}_3(vb, wb, s, t) = \mathsf{tckbo}(vb, wb, s, t)$ *and* $\mathsf{tckbolex}_3(vb, wb, ss, ts) = \mathsf{tckbolex}(vb, wb, ss, ts)$*.*

PROOF Simultaneous induction on $|s| + |t|$ and $|ss| + |ts|$. Lemma 4.8 covers the cases where $s =_{\mathsf{t}} t$ and $ss =_{\mathsf{tl}} ts$. Hence, it suffices to consider the remaining cases.

If s and t are different variables then the variable balances are updated accordingly and N is determined for the ordering relation. As φ maps all variables to μ, wb remains unchanged. If one term is a variable and the other is not then calcVWBc modifies the balances for the nonvariable terms. Furthermore, it tests whether the variable occurs in the term which determines the value of res. Finally, the balances are updated to take the variable into account. It remains the case where s and t are both nonvariable terms. If both top-symbols are identical then $\mathsf{tckbolex}_3$ is called which by induction hypothesis returns the correct result for the comparison of the arguments. Otherwise, tckbo'_3 updates the balances for the arguments by calling $\mathsf{calcVWB}_{\mathsf{tl}}$. Handling the weights of the top-symbols is the duty of tckbo_3. Variables $G\text{-}or\text{-}N$ and $L\text{-}or\text{-}N$ record whether or not the variable balances approve a possible G or L result. The **if**-**elif**-expression then simply follows the definition of KBO and considers first the weights, then the precedence, and finally the result of the lexicographic comparison.

If ss and ts are not identical then, after considering some possibly empty prefix, $\mathsf{tckbolex}_3$ calls tckbo_3 with $vb = \vec{0}$, $wb = 0$, and two different terms. Hence, by induction hypothesis, $res \neq \mathsf{E}$ and the balances are updated accordingly. □

With the help of tckbo_3 it is easy to compute ckbo_3 which is mainly a wrapper that handles the allocation and deallocation of the array for the variable balances.

$$
\begin{aligned}
\mathsf{ckbo}_3(s,t) \;=\; \textbf{let} \quad & vb = \mathsf{newVB}(\langle\rangle) \\
& \langle vb', wb, res\rangle = \mathsf{tckbo}_3(vb, 0, s, t) \\
& \langle\rangle = \mathsf{freeVB}(vb') \\
\textbf{in} \quad & res
\end{aligned}
$$

THEOREM 4.10 *Let s and t be well-formed* Term*s. Then* $\mathsf{ckbo}_3(s,t) = \mathsf{ckbo}_2(s,t)$*. The worst-case running time of* $\mathsf{ckbo}_3(s,t)$ *is* $O(KN)$ *where* $K = |\mathcal{V}_{\mathrm{fin}}|$ *and* $N = |s| + |t|$*.*

PROOF Function ckbo_3 calls newVB, tckbo_3, and freeVB. Lemma 4.9 ensures the correctness of ckbo_3. The costs for newVB and freeVB are $O(K)$. During the evaluation of $\mathsf{tckbo}_3(vb, wb, s, t)$ each symbol in s and t is considered once. Except functions noNeg and noPos all operations need constant time. Functions noNeg and noPos need $O(K)$ time for each invocation. In the worst case, both functions are called $N/2$ times. Hence, $\mathsf{tckbo}_3(vb, wb, s, t)$ needs at most $O(KN)$ steps. Therefore, the worst-case running time of $\mathsf{ckbo}_3(s,t)$ is $O(KN)$. □

Revisiting Example 4.1 again, we can see in Figure 4.8 that the running time of KBO$_3$, the C-implementation of kbo_3, depends linearly on the size of its arguments. It is also considerably faster than KBO$_2$ for these test cases. (Note the different scale of Figures 4.7 and 4.8). The slope of the lines is linear with K.

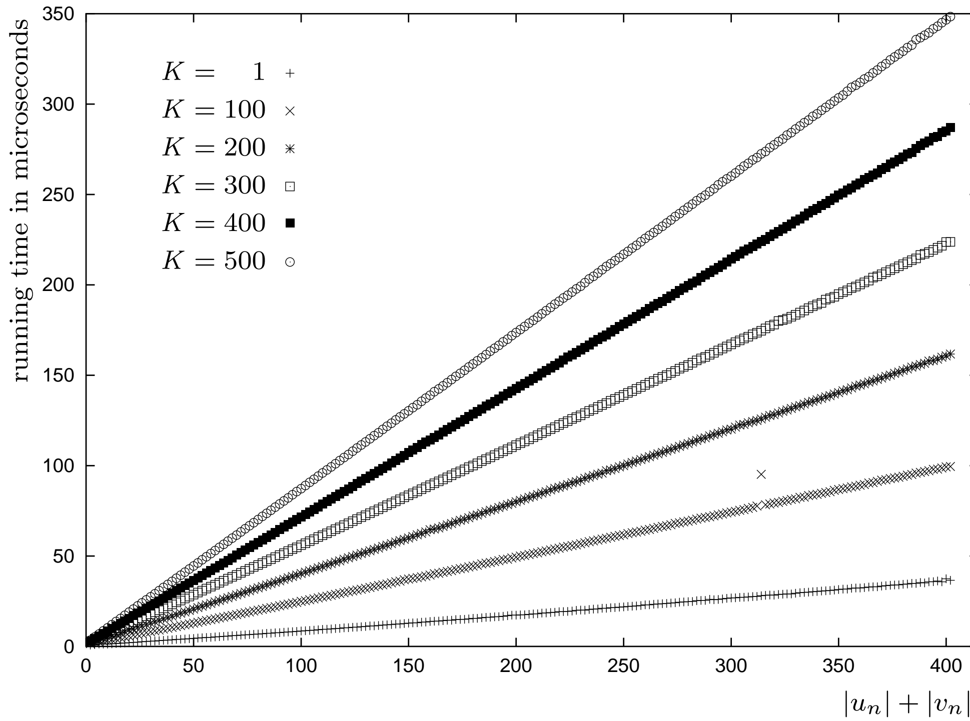


Figure 4.8: Time needed to evaluate $\textsc{kbo}_3(u_n, v_n)$ with $n = 1, \ldots, 200$ and six different values of K (cf. Example 4.1)

Optimizing the variable test

By using the tupling strategy we have successfully avoided the quadratic behavior. However, as can be seen in Figure 4.8, the running time of $\textsc{kbo}_3$ still depends noticeably on K, the number of different variables in the prover's state. Each invocation of noNeg and noPos tests for each of the K variables its balance, which apparently is in $O(K)$. The main idea to transform both operations into tests with constant costs is to keep counters for the number of positive variable balances and for the number of negative variable balances. Let getPC and getNC return the values of these counters. Then we can implement noNeg and noPos in the following way:

$$\begin{array}{rcl} \mathsf{noNeg} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noNeg}(vb) & = & \mathbf{let}\ \langle vb', n\rangle = \mathsf{getNC}(vb) \\ & & \mathbf{in}\ \ \langle vb', n = 0\rangle \end{array} \qquad \begin{array}{rcl} \mathsf{noPos} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Bool} \\ \mathsf{noPos}(vb) & = & \mathbf{let}\ \langle vb', n\rangle = \mathsf{getPC}(vb) \\ & & \mathbf{in}\ \ \langle vb', n = 0\rangle \end{array}$$

A nice effect of this change is that functions noNeg and noPos are now independent from the representation of VarBal. This knowledge is hidden in getNC and getPC.

We keep the two counters in two extra entries of the array that represents the variable balances. We use entry 0 to count the negative variable balances and entry $K + 1$ to count the positive variable balances. We therefore have to change function newVB:

$$\begin{array}{rcl} \mathsf{newVB} & : & \mathbf{1} \to \mathsf{VarBal} \\ \mathsf{newVB}(\langle\rangle) & = & \mathsf{newArray}(\mathsf{Int}, 0, K + 1) \end{array}$$

The following functions allow us to read, to increment, and to decrement the two counters:

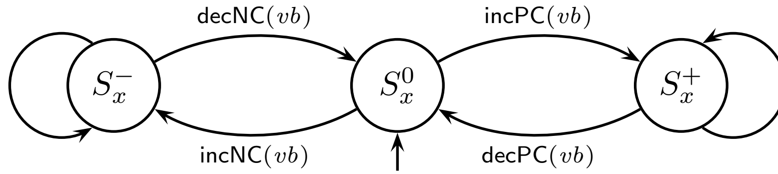
$$\begin{array}{rcl}
\mathsf{getNC} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Int} \\
\mathsf{getNC}(vb) & = & \mathsf{read}(vb, 0)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{getPC} & : & \mathsf{VarBal} \to \mathsf{VarBal} \times \mathsf{Int} \\
\mathsf{getPC}(vb) & = & \mathsf{read}(vb, K+1)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{incNC} & : & \mathsf{VarBal} \to \mathsf{VarBal} \\
\mathsf{incNC}(vb) & = & \mathbf{let}\ \langle vb', n\rangle = \mathsf{read}(vb, 0) \\
 & & \mathbf{in}\ \ \mathsf{update}(vb', i, n+1)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{incPC} & : & \mathsf{VarBal} \to \mathsf{VarBal} \\
\mathsf{incPC}(vb) & = & \mathbf{let}\ \langle vb', n\rangle = \mathsf{read}(vb, K+1) \\
 & & \mathbf{in}\ \ \mathsf{update}(vb', i, n+1)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{decNC} & : & \mathsf{VarBal} \to \mathsf{VarBal} \\
\mathsf{decNC}(vb) & = & \mathbf{let}\ \langle vb', n\rangle = \mathsf{read}(vb, 0) \\
 & & \mathbf{in}\ \ \mathsf{update}(vb', i, n-1)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{decPC} & : & \mathsf{VarBal} \to \mathsf{VarBal} \\
\mathsf{decPC}(vb) & = & \mathbf{let}\ \langle vb', n\rangle = \mathsf{read}(vb, K+1) \\
 & & \mathbf{in}\ \ \mathsf{update}(vb', i, n-1)
\end{array}$$

It is clear from these definitions that functions getNC, getPC, incNC, decNC, incPC, and decPC all need constant time.

We modify the counters each time the variable balance for a variable x leaves or reaches 0. Thus, we treat the variable balance for x as the following finite automaton:



Functions inc and dec are then defined accordingly:

$$\begin{array}{rcl}
\mathsf{inc} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\
\mathsf{inc}(vb, x) & = & \mathbf{let}\ \quad i = \mathsf{index}(x) \\
 & & \quad \langle vb', n\rangle = \mathsf{read}(vb, i) \\
 & & \quad vb'' = \mathsf{update}(vb', i, n+1) \\
 & & \mathbf{in}\ \mathbf{if}\ \ n = 0\ \ \mathbf{then}\ \mathsf{incPC}(vb'') \\
 & & \quad \mathbf{elif}\ n = -1\ \mathbf{then}\ \mathsf{decNC}(vb'') \\
 & & \quad \mathbf{else}\ vb''
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{dec} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\
\mathsf{dec}(vb, x) & = & \mathbf{let}\ \quad i = \mathsf{index}(x) \\
 & & \quad \langle vb', n\rangle = \mathsf{read}(vb, i) \\
 & & \quad vb'' = \mathsf{update}(vb', i, n-1) \\
 & & \mathbf{in}\ \mathbf{if}\ \ n = 0\ \mathbf{then}\ \mathsf{incNC}(vb'') \\
 & & \quad \mathbf{elif}\ n = 1\ \mathbf{then}\ \mathsf{decPC}(vb'') \\
 & & \quad \mathbf{else}\ vb''
\end{array}$$

Keeping the two counters has a profound influence on the running time. As getNC and getPC both need constant time, functions noNeg and noPos are both in $O(1)$. Functions inc and dec need more time than before, but are still in $O(1)$. Hence, we get the following result for function ckbo_4 which differs from ckbo_3 by using the counter-based variable tests:

THEOREM 4.11 *Let s and t be well-formed* Term*s. Then* $\mathsf{ckbo}_4(s,t) = \mathsf{ckbo}_3(s,t)$*. The worst-case running time of* $\mathsf{ckbo}_4(s,t)$ *is* $O(K+N)$ *where* $K = |\mathcal{V}_{\text{fin}}|$ *and* $N = |s|+|t|$.

PROOF It is easy to see that the counter-based variable test is equivalent to the previous one. For determining the costs, recall that function ckbo_4 invokes newVB,

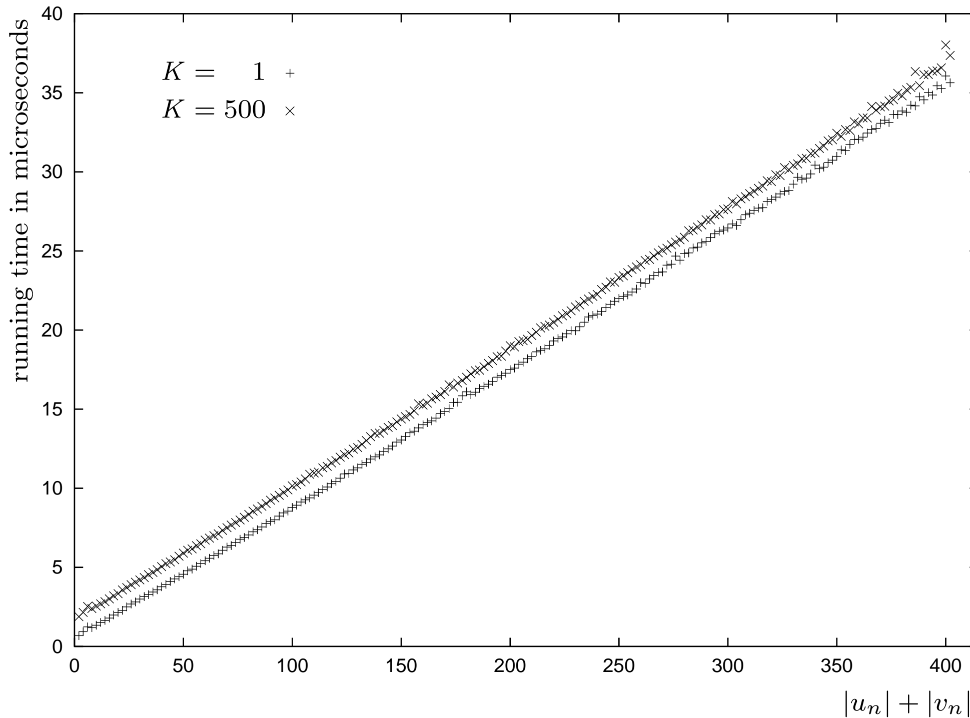


Figure 4.9: Time needed to evaluate $\text{KBO}_4(u_n, v_n)$ with $n = 1, \ldots, 200$ and two different values of K (cf. Example 4.1)

tckbo_4, and freeVB. The costs for newVB and freeVB are $O(K)$. During the evaluation of $\mathsf{tckbo}_4(vb, wb, s, t)$ each symbol in s and t is considered once. Including functions noNeg and noPos the operations performed for each symbol need constant time. Hence, $\mathsf{tckbo}_4(vb, wb, s, t)$ needs $O(N)$ time. Therefore, the worst-case running time of $\mathsf{ckbo}_4(s, t)$ is $O(K + N)$. □

We denote the C-implementation of kbo_4 with KBO_4. Revisiting Example 4.1 again, its dependency on K shows as a small constant offset in the two graphs depicted in Figure 4.9. (For clarity reasons, we have omitted the data for the other values of K). For $K = 500$, KBO_4 is also considerably faster than KBO_3, whereas for $K = 1$ the differences are tiny, as can be seen by comparison with Figure 4.8.

One final modification

The optimized handling of the variable test has successfully avoided the dependency on K of functions noNeg and noPos. However, functions newVB and freeVB may still depend on K. The aim of our final modification is to change that. As a result, the evaluation of $\mathsf{ckbo}_5(s, t)$ is independent of the number of variables in the prover's state. The practical improvements we can expect are rather small, as Figure 4.9 indicates. The real importance of this result is that it is asymptotically optimal: the running time of $\mathsf{ckbo}_5(s, t)$ solely depends on $|s|$ and $|t|$, the size of its actual arguments.

A general observation with the theorem prover WALDMEISTER is that calls to the ordering occur much more frequently than changes of K. We therefore make the follow-

ing modification: the memory for the variable balances is not allocated and deallocated for each invocation of ckbo_5. Instead, this memory is handled outside the ordering in some global variable *globalVB*. If $\mathcal{V}_{\text{fin}}$ changes then the size of *globalVB* is adjusted accordingly.[11] This is not the duty of the module implementing the ordering. Therefore, we have to establish some invariant about the contents of *globalVB*. We assume that all entries of *globalVB* are zero at the invocation of ckbo_5. Hence, we have to ensure that this holds true when ckbo finishes.

The task of function freeVB is therefore to zero out all entries that are modified during the evaluation of $\mathsf{tckbo}_5(vb, wb, s, t)$. To do this independently of K, we have to record the modified entries during the computation. Therefore, we change the representation of variable balances. Data type VarBal is now an array of pairs of integers with indices ranging from 0 to $K + 1$. Of each entry, the first component keeps the variable balance as before. With the second component we establish a singly-linked list of the modified entries. To read and modify the two components of an entry independently we use the following functions:

$$
\begin{array}{rcl}
\mathsf{read}_1 & : & \mathsf{VarBal}\ \mathsf{Nat} \to \mathsf{VarBal} \times \mathsf{Int} \\
\mathsf{read}_1(vb, i) & = & \textbf{let}\ \langle vb', \langle n_1, n_2\rangle\rangle = \mathsf{read}(vb, i) \\
 & & \textbf{in}\ \ \langle vb', n_1\rangle \\
\\
\mathsf{update}_1 & : & \mathsf{VarBal}\ \mathsf{Nat}\ \mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{update}_1(vb, i, n) & = & \textbf{let}\ \langle vb', \langle n_1, n_2\rangle\rangle = \mathsf{read}(vb, i) \\
 & & \textbf{in}\ \ \mathsf{update}(vb', \langle n, n_2\rangle) \\
\\
\mathsf{read}_2 & : & \mathsf{VarBal}\ \mathsf{Nat} \to \mathsf{VarBal} \times \mathsf{Int} \\
\mathsf{read}_2(vb, i) & = & \textbf{let}\ \langle vb', \langle n_1, n_2\rangle\rangle = \mathsf{read}(vb, i) \\
 & & \textbf{in}\ \ \langle vb', n_2\rangle \\
\\
\mathsf{update}_2 & : & \mathsf{VarBal}\ \mathsf{Nat}\ \mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{update}_2(vb, i, n) & = & \textbf{let}\ \langle vb', \langle n_1, n_2\rangle\rangle = \mathsf{read}(vb, i) \\
 & & \textbf{in}\ \ \mathsf{update}(vb', \langle n_1, n\rangle)
\end{array}
$$

The new versions of functions getNC, getPC, incNC, decNC, incPC, and decPC are identical to the old ones, except that read is replaced by read_1 and update is replaced by update_1. Hence, their running times remain constant. Functions inc and dec are changed analogously. In addition, they call function record to indicate that the entry is changed:

[11] It is not strictly necessary to keep a global array for that purpose. Many theorem provers keep for each variable a structure for recording its name, sort, etc. Given a certain variable, the corresponding structure is usually quickly accessible. We could therefore avoid *globalVB* by storing its contents in these structures. It is easy to adapt the simpler array-based formulation we have chosen to such a setting.

$$
\begin{array}{rcl}
\mathsf{inc} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\
\mathsf{inc}(vb, x) & = & \textbf{let} \quad i = \mathsf{index}(x) \\
& & \quad \langle vb', n\rangle = \mathsf{read}_1(vb, i) \\
& & \quad vb'' = \mathsf{update}_1(vb', i, n+1) \\
& & \quad vb''' = \mathsf{record}(vb'', i) \\
& & \textbf{in}\ \textbf{if}\ n = 0\ \textbf{then}\ \mathsf{incPC}(vb''') \\
& & \quad \textbf{elif}\ n = -1\ \textbf{then}\ \mathsf{decNC}(vb''') \\
& & \quad \textbf{else}\ vb'''
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{dec} & : & \mathsf{VarBal}\,\mathsf{Vid} \to \mathsf{VarBal} \\
\mathsf{dec}(vb, x) & = & \textbf{let} \quad i = \mathsf{index}(x) \\
& & \quad \langle vb', n\rangle = \mathsf{read}_1(vb, i) \\
& & \quad vb'' = \mathsf{update}_1(vb', i, n-1) \\
& & \quad vb''' = \mathsf{record}(vb'', i) \\
& & \textbf{in}\ \textbf{if}\ n = 0\ \textbf{then}\ \mathsf{incNC}(vb''') \\
& & \quad \textbf{elif}\ n = 1\ \textbf{then}\ \mathsf{decPC}(vb''') \\
& & \quad \textbf{else}\ vb'''
\end{array}
$$

The singly-linked list is handled by the following functions. We use the value $K+1$ as sentinel for the end of the list. The anchor of the list is the second component of the zeroth array entry. We assume that variable $globalVB$ contains the result of $\mathsf{newArray}(\mathsf{Int} \times \mathsf{Int}, 0, K+1)$.

$$
\begin{array}{rcl}
\mathsf{newVB} & : & \mathbf{1} \to \mathsf{VarBal} \\
\mathsf{newVB}(\langle\rangle) & = & \textbf{let}\ vb = globalVB \\
& & \quad vb' = \mathsf{update}_2(vb, 0, K+1) \\
& & \textbf{in}\ vb' \\
\mathsf{freeVB} & : & \mathsf{VarBal} \to \mathbf{1} \\
\mathsf{freeVB}(vb) & = & \textbf{let}\ vb' = \mathsf{clear}(vb, 0) \\
& & \textbf{in}\ \langle\rangle \\
\mathsf{clear} & : & \mathsf{VarBal}\ \mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{clear}(vb, i) & = & \textbf{let}\ \langle vb', i'\rangle = \mathsf{read}_2(vb, i) \\
& & \quad vb'' = \mathsf{update}(vb', i, \langle 0, 0\rangle) \\
& & \textbf{in}\ \textbf{if}\ i = K+1\ \textbf{then}\ vb'' \\
& & \quad \textbf{else}\ \mathsf{clear}(vb'', i') \\
\mathsf{record} & : & \mathsf{VarBal}\ \mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{record}(vb, i) & = & \textbf{let}\ \langle vb', i'\rangle = \mathsf{read}_2(vb, i) \\
& & \textbf{in}\ \textbf{if}\ i' = 0\ \textbf{then}\ \mathsf{record}'(vb', i) \\
& & \quad \textbf{else}\ vb' \\
\mathsf{record}' & : & \mathsf{VarBal}\ \mathsf{Int} \to \mathsf{VarBal} \\
\mathsf{record}'(vb, i) & = & \textbf{let}\ \langle vb', i'\rangle = \mathsf{read}_2(vb, 0) \\
& & \quad vb'' = \mathsf{update}_2(vb', i, i') \\
& & \quad vb''' = \mathsf{update}_2(vb'', 0, i) \\
& & \textbf{in}\ vb'''
\end{array}
$$

Functions newVB and record need constant time. Hence, functions inc and dec need constant time, too. The running time of freeVB is determined by the length of the list. If there are M different variables in s and t then function clear overwrites $M+2$ entries of the array: In addition to the entries representing variable balances the two entries at position 0 and $K+1$ which represent the counters. Let $N = |s| + |t|$. Then $M \leq N$. This leads to the desired result for function ckbo_5 which uses the new versions of the functions handling the variable balances.

THEOREM 4.12 *Let s and t be well-formed* Terms*. Then* $\mathsf{ckbo}_5(s,t) = \mathsf{ckbo}_4(s,t)$*. The worst-case running time of* $\mathsf{ckbo}_5(s,t)$ *is* $O(|s| + |t|)$*.*

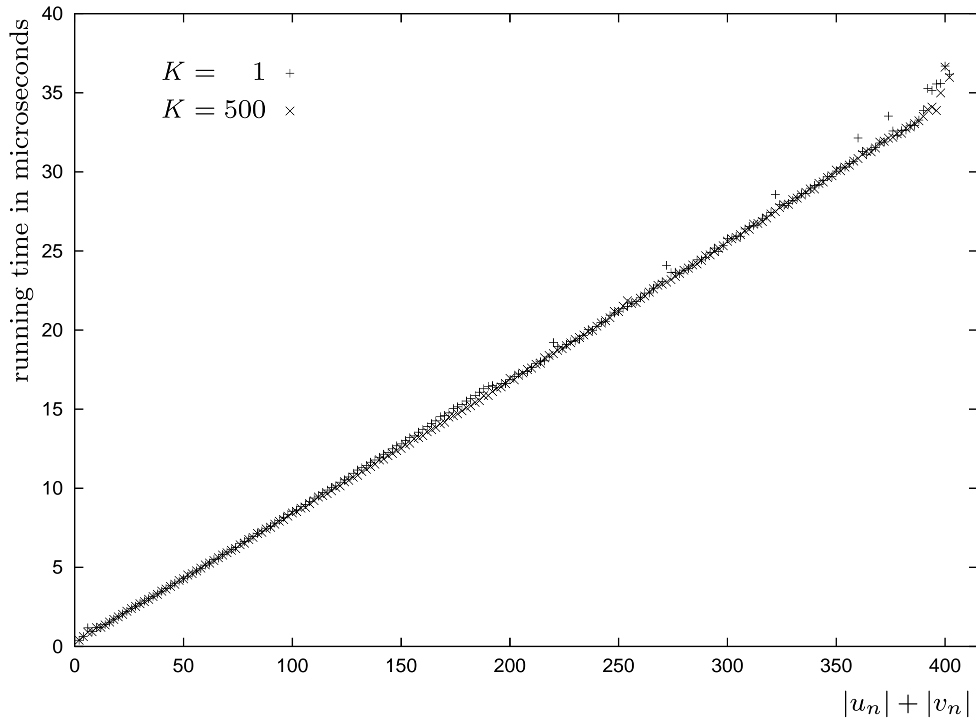


Figure 4.10: Time needed to evaluate KBO$_5(u_n, v_n)$ with $n = 1, \ldots, 200$ and two different values of K (cf. Example 4.1)

PROOF The different memory management does not affect correctness as all modified entries are zeroed out by clear. Concerning the costs, function ckbo$_5$ calls besides tckbo$_5$ the new versions of newVB and freeVB. The costs for newVB are $O(1)$. The costs for freeVB are $O(M)$ where $M = |\mathrm{Var}(s,t)| \leq |s| + |t|$. Except calling the new versions of the auxiliary functions, function tckbo$_5$ is identical to function tckbo$_4$. As these auxiliary functions all need constant time, evaluating tckbo$_5(vb, wb, s, t)$ needs $O(|s|+|t|)$ time. Therefore, the worst-case running time of ckbo$_5(s,t)$ is $O(|s|+|t|)$. □

KBO$_5$ is our C-implementation of kbo$_5$. Figure 4.10 depicts the running times of KBO$_5$ for the test cases of Example 4.1 and two different values of K. The two graphs are practically indistinguishable, which indicates that the running time of KBO$_5$ does not depend on K. The differences to KBO$_4$ are, however, rather small, cf. Figure 4.9.

Experimental evaluation

During the development of the different versions of kbo we presented experimental data for a well chosen worst-case example to illustrate the different asymptotic running time behavior. Unfortunately, such experiments tell us nothing about the improvements we can expect in practice: the terms tested for their ordering relations in real proof attempts do not follow such simple patterns.

Similarly to LPO, we performed two kinds of experiments: For five individual examples we measured the time for the calls to the ordering until WALDMEISTER finished successfully. The examples belong to different domains of TPTP [SS98]. Furthermore,

Table 4.3: Time (in seconds) needed for ordering comparisons

problem	number of calls to ordering	time needed by ordering					time needed by other operations
		KBO$_1$	KBO$_2$	KBO$_3$	KBO$_4$	KBO$_5$	
BOO031-1	455 754	1.500	0.613	0.477	0.531	0.401	34.554
GRP179-1	436 624	0.881	0.527	0.433	0.463	0.324	36.394
LCL109-2	106 160	0.337	0.166	0.107	0.121	0.081	12.945
LAT085-1	195 780	1.370	0.356	0.258	0.287	0.185	16.717
ROB006-1	511 148	2.544	0.894	0.757	0.767	0.590	13.185
TPTP$_{10}$	64 151 129	307.800	114.400	85.400	97.350	77.660	1 402.300

we made measurements for all 722 UEQ-problems of TPTP-2.7.0. For that, we first ran WALDMEISTER using KBO$_1$ with a time limit of 10 seconds and recorded how often the ordering routines were called. We then performed the actual measurements by running the prover with the different versions KBO$_i$ and by aborting the run after the corresponding number of calls is reached. We use TPTP$_{10}$ to refer to the summarized results.

As we can learn from Table 4.3, the optimizations usually improve the time needed for the ordering comparisons. However, the improvements are more moderate than for LPO. This is not surprising as kbo_1 is already polynomial. Hence, even KBO$_1$ requires only a fraction of the time that the other operations of the prover need. An interesting observation is that KBO$_4$ usually needs more time than KBO$_3$. This means that in these experiments the counter-based variable test is inferior to simply checking the variable balances individually. The reason is that in WALDMEISTER the number of different variables is kept as small as possible. Therefore, we can observe rather small values of K. For the runs of TPTP$_{10}$ the average value of K is 9.5, and for only 13 of the 722 problems it is greater than twenty. Therefore, only few comparisons can be saved, whereas for each invocation of inc and dec additional comparisons are necessary. The potential of the optimization lies in cases where K is greater than the number of variables occurring in the terms. In the test runs of TPTP$_{10}$ these cases are too rare to lead to speed-ups. Considering the small average value of K, we have to attribute the improvements of KBO$_5$ to the significantly reduced number of allocations and deallocations. The linear dependency on K of KBO$_4$ does not seem relevant for such small values of K.

4.4 Related work and implementation status

With Snyder [Sny93] we share the finding that the number of publications about the efficient implementation of orderings is really small compared to the number of publications about the definition of new orderings. The thesis of Steinbach contains a

short overview about the time complexities of orderings [Ste94, Chap. 6.2]. It seems to be agreed in the literature that the standard way to achieve a polynomial version of LPO and related orderings is via dynamic programming [KNS85]. Snyder devised an $O(n \cdot \log n)$ algorithm for comparing ground terms with LPO based on total precedences [Sny93].

In [Wei01] Weidenbach acknowledges that the top-down implementation of LPO may result in exponential behavior. He then goes on and compares the top-down implementation with an implementation of [Sny93] and concludes that the latter gives no advantages in practice. Inspection of the source code of SPASS [WBH+02] shows that it contains a top-down implementation of the *Recursive Path Ordering with Status* (RPOS). This is a well-known generalization of LPO, where each function symbol has assigned to it a status (left-to-right, right-to-left, or multiset). If all function symbols have status left-to-right, we get LPO as a special case. Then the RPOS-implementation of SPASS behaves like lpo_5. That this is actually a polynomial version is not mentioned.

Analyzing the source code of other systems that participated in CASC, the system competition of automated theorem provers affiliated with the Conference on Automated Deduction [S+], we find no implementation of the dynamic programming algorithm. Snyder's method seems to be implemented in SPASS only. In OTTER [McC05a] we find an implementation of RPOS which specializes essentially to lpo_4 if all functions symbols have status left-to-right. WALDMEISTER's original LPO is an elaborate version of lpo_6. Several costly pretest are used in fear of exponential behavior. The straightforward implementation of lpo_4 is more than three times faster. The other systems that we analyzed use a version similar to lpo_1 or lpo_2, sometimes enriched with optimizations for special cases. Often some form of caching (of different implementation quality) is used to avoid exponential behavior.[12]

We are unaware of any work discussing the efficient implementation of the basic KBO test ([RV04] has a different aim). Steinbach considers the KBOS, a variant of KBO that assigns each operator a status. He shows that it can be evaluated in $O(|s| \cdot |t|)$ time. Weidenbach suggests to implement the KBO straight after its definition [Wei01].

Compared to LPO/RPOS, fewer CASC systems implement the KBO. Some systems use simple weight-based orderings instead. Typically, a recursive variant similar to kbo_2 is employed performing the variable test with the help of some data structure. We find the use of arrays, hash tables, and association lists. An important variation is to delay the variable test, that is, it is only performed after the tests belonging to part (2) of Definition 4.2 have been checked successfully. This helps to reduce the costs for the variable test. A different approach is used in the Vampire system where the variable test and the weight-based test are combined by using (linear) polynomials. Recurring tests with two terms under different substitutions are optimized by using a specialized partial evaluation technique (see [RV04]). All implementations we investigated show quadratic worst-case behavior.

[12] After reading a preliminary version of this work, Schulz replaced his old LPO-implementation with lpo_4 in his theorem prover E [Sch02]. Some TPTP-problems show an overall speedup of 20. (Personal communication, May 2004.)

4.5 Conclusions

Starting from "obviously correct", but inefficient variants, we developed step-by-step efficient implementations of LPO and KBO. For LPO, from an initially exponential algorithm we derive a quadratic version, mainly by the use of ordering specific knowledge. For KBO, from an initially quadratic algorithm, which depends furthermore on the number of variables in the system K, we derive a linear one which is independent of K. Here, we use standard program transformation techniques and appropriate data structures. To our knowledge, this is the first linear implementation of KBO.

At the ESFOR workshop in 2004, we presented the material concerning LPO [Löc04b]. This spawned a lively discussion which revealed that on the one hand for several researchers it was part of folklore that LPO can be implemented in polynomial time without the use of additional data structures.[13] On the other hand, several known implementors did not know the complexities of their implementations of LPO. Furthermore, several researchers did not even know that LPO can be implemented in polynomial time.[14] Although several systems contain code similar to $\mathsf{lpo}_{\mathsf{R5}}$ or clpo_6, no one could give a reference to the literature for these algorithms. It was consensus that implementing an automated theorem prover is a complex task and that an implementor has to tackle at least two or three dozens of subproblems of at least the difficulty of LPO. Resources are too limited to treat all of them with the necessary care. Our work has increased the awareness that too much knowledge is considered folklore or is buried deeply in the code of other systems.

We have not yet presented the material concerning KBO to a wider audience. However, several developers of CASC-systems were surprised to hear that KBO can be implemented in linear time.

We concentrated on the basic ordering test $s \succ t$. An important extension is to determine for terms s and t and substitution σ whether $\sigma(s) \succ \sigma(t)$ holds. By passing σ as an additional parameter in the recursive calls and looking up bindings of variables if necessary, we can improve considerably on explicitly constructing $\sigma(s)$ and $\sigma(t)$ before calling the basic ordering test. In some situations s and t are fixed and σ is varying. Then, we can achieve further improvements by preprocessing s and t. Riazanov and Voronkov considered this and similar problems for KBO [RV04]. For some uses in a prover an approximation $\succ^{\mathrm{a}}$ of the ordering would suffice. This means that $s \succ^{\mathrm{a}} t$ implies $s \succ t$, but from $s \not\succ^{\mathrm{a}} t$ we can deduce nothing. The weakest (and cheapest) approximation would return false for all terms s and t, the strongest would be $\succ$ itself. Does there exist a sub-quadratic approximation of LPO that is reasonably precise?

To describe the different variants of the orderings we used a small algebraic specification language to keep the presentation concise. The translation of the different specifications into the input language of the inductive theorem prover QuodLibet [AKSSW03] was straightforward. We later realized that other inductive provers have problems with mutually recursive functions and partiality (e. g., lex_1 is only partially

---

[13] According to Baader's statement at the ESFOR workshop, Exercise 5.25 of the textbook [BN98] does not aim at the use of dynamic programming techniques. When this exercise is handed out in his courses on term rewriting, usually one or two students come up with a version similar to $\mathsf{lpo}_{\mathsf{R5}}$ or clpo_6.

[14] This explains our findings when we analyzed the source code of different systems.

defined). Trying to prove properties of the different versions with QuodLibet revealed implicit assumptions in the code. For example, to show $\mathsf{lpo}_1(s,t) = \mathsf{lpo}_2(s,t)$ we need that $\mathsf{well}(s) = \mathsf{well}(t) = \mathsf{true}$, i. e., that the arguments are well-formed terms. Our examples spawned some interest in the development of tactics that are specialized in the support of mutually recursive functions. This is ongoing work.

For developing the different versions we used the paradigm of program transformations. This helped not only to focus on the essential ideas, but also to prevent errors in the implementation – even by doing it by hand without the support of a dedicated system. In our C translations of all LPO-variants considered we discovered *three* bugs, which were all caught easily by comparing the different versions during test-runs. One bug occurred because of a copying error, the other two by replacing recursive calls with iteration constructs. Considering KBO, most bugs occurred in the development of $\textsc{kbo}_3$. This indicates, that the step between kbo_2 and kbo_3 is too wide to do without machine support. Nevertheless, this low number of errors is far better than what is usually achieved by traditional coding practice, especially when we take into account the intricacy of the algorithms. This experience suggests to use this two-level development approach for other subtasks in a prover, especially when they need a significant amount of the running time and an efficient implementation is not obvious.

5 Ordering constraints

In the last 15 years, ordering constraints have been developed to become an important means in automated deduction. For example, modern treatments of the superposition calculus use a constrained based formulation [GN01, NR01]. Our use or ordering constraints is inside some of the redundancy criteria which are the topic of Chapter 6. For this application they turn out to be an indispensable tool. With this more localized use, we sacrifice some of the theoretical elegance that constraint inheritance based calculi have. However, we avoid not only their theoretical problems (e. g., the simplification relation of [KKR90] is undecidable because of the kind of constraints needed [CT97]), but also the more practical ones, which prevented the use of constraints in high-performance systems.

Ordering constraints allow us to precisely formulate restrictions for the set of (ground) instances of a syntactic expression. For example, ordered rewriting with an unoriented equation $u = v$ uses orientable instances for simplification (i. e., with $\sigma(u) \succ \sigma(v)$). With ordering constraints we can conveniently describe the set of reducing instances by substitutions *satisfying* the constraint $u > v$. Conversely, substitutions describing (ground) instances that are not reducible in this direction satisfy the constraint $v \geq u$, as the reduction orderings we use are total on ground terms. In Section 5.1 we define the syntax of constraints and the notion of satisfiability we use.

The most important algorithmic problem with ordering constraints is to decide whether a given constraint is satisfiable. Although this problem inherently depends on the underlying ordering scheme, the general case can easily be shown to be NP-hard by reduction of 3-SAT (see end of Section 5.1). NP-completeness was shown for LPO in [Nie93b] and for KBO in [KV01]. It is therefore appropriate to develop sufficient tests for the unsatisfiability of constraints in addition to searching for decision procedures.

First we concentrate on ordering specific tests: In Section 5.2 we describe a decision procedure for LPO devised in [NR02]. In Section 5.3 we describe several sufficient tests for KBO which are all based on reduction schemes to linear arithmetic, the quantifier-free fragment of Presburger arithmetic, like the decision procedure of [KV01]. We omit the latter, however, because it is very complicated to describe and difficult to implement. Then we describe a generic sufficient test with polynomial runtime requirements in Section 5.4. It is suitable for arbitrary ground reduction orderings and can be extended to handle specific orderings. We continue in Section 5.5 with an experimental comparison of the different tests considering running time and accuracy. With Section 5.6 we conclude.

5.1 Syntax and satisfiability of ordering constraints

In principle, ordering constraints are quantifier-free formulas over the predicate symbols $>$, $\geq$, and $=$, which are interpreted over the ground terms with respect to the underlying reduction ordering $\succ$ and syntactic identity $\equiv$.

DEFINITION 5.1 Atomic constraints *are either* inequality constraints *of the form* $s > t$ *or* $s \geq t$, *or* equality constraints *of the form* $s = t$, *where* $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. *The set of* ordering constraints *is the smallest set that contains the atomic constraints and is closed under conjunction and disjunction of constraints, that is, whenever* $C_1, \ldots, C_n$ *are constraints then* $C_1 \wedge \ldots \wedge C_n$ *and* $C_1 \vee \ldots \vee C_n$ *are constraints. The* trivial constraints *are* $\top$ *(for the empty conjunction) and* $\bot$ *(for the empty disjunction).*

We regard equality constraints as symmetric and do not distinguish $s = t$ and $t = s$. Occasionally we will use an ordering $\sqsupset$ on constraint symbols. We have $> \sqsupset \geq \sqsupset =$. Because $\succ$ is total on ground terms, there is no need to consider negation in formulas describing ground instances: $\neg(s > t)$ is equivalent to $t \geq s$, $\neg(s \geq t)$ is equivalent to $t > s$, and $\neg(s = t)$ is equivalent to $(s > t) \vee (t > s)$. This is clarified in the following definition of satisfiability of ordering constraints which is based on substitutions.

DEFINITION 5.2 *Let* $\succ$ *be the underlying ground reduction ordering. A substitution* σ satisfies *the atomic constraints* $s = t$, $s > t$, *or* $s \geq t$ *iff* $\sigma(s) \equiv \sigma(t)$, $\sigma(s) \succ \sigma(t)$, *or* $\sigma(s) \succcurlyeq \sigma(t)$, *respectively. A conjunction* $C_1 \wedge \ldots \wedge C_n$ *is satisfied by* σ *iff all* C_i *are satisfied by* σ, *a disjunction* $C_1 \vee \ldots \vee C_n$ *is satisfied by* σ *iff at least one* C_i *is satisfied by* σ. *A* solution *of* C *is a substitution that satisfies* C. *We write* $\mathrm{Sol}(C)$ *for the set of solutions of* C. *The set of* ground solutions $\mathrm{GSol}(C)$ *is given by* $\mathrm{GSol}(C) = \mathrm{Sol}(C) \cap \mathrm{GSub}(C)$. *A constraint* C *is* satisfiable *with respect to* $\succ$ *iff* $\mathrm{GSol}(C) \neq \varnothing$, *that is, there is at least one ground substitution that satisfies* C *with respect to* $\succ$.

By this definition, an equality constraint $s = t$ is satisfied exactly by its unifiers. Furthermore, $\top$ is satisfied by any substitution and is the neutral element of conjunction (i.e., $\mathrm{Sol}(C \wedge \top) = \mathrm{Sol}(C)$), whereas $\bot$ is trivially unsatisfiable and the neutral element of disjunction (i.e., $\mathrm{Sol}(C \vee \bot) = \mathrm{Sol}(C)$). More generally, we have $\mathrm{Sol}(C_1 \wedge \ldots \wedge C_n) = \mathrm{Sol}(C_1) \cap \cdots \cap \mathrm{Sol}(C_n)$ for conjunctions and analogously $\mathrm{Sol}(C_1 \vee \ldots \vee C_n) = \mathrm{Sol}(C_1) \cup \cdots \cup \mathrm{Sol}(C_n)$ for disjunctions.

Considering ground solutions, a corresponding property holds for conjunctions but not for disjunctions. Because of the domain restrictions, $\mathrm{GSol}(C_1) \subseteq \mathrm{GSol}(C_1 \vee C_2)$ does not hold. Consider for example $C_1 \equiv x > a$ and $C_2 \equiv y \geq a$. For a ground solution $\sigma \in \mathrm{GSol}(C_1)$ it is required that $x \in \mathrm{dom}(\sigma)$, but not that $y \in \mathrm{dom}(\sigma)$. Hence, $\sigma \in \mathrm{Sol}(C_1 \vee C_2)$ but not necessarily $\sigma \in \mathrm{GSol}(C_1 \vee C_2)$. However, for any constraint C it is easy to extend a solution $\sigma \in \mathrm{Sol}(C)$ to a ground solution in the following way: Let c be an arbitrary constant, $\sigma_c = \{x \mapsto c \mid x \in \mathrm{Var}(C) \cup \mathcal{I}(\sigma)\}$, and $\sigma_g = \sigma_c \circ \sigma$. Then $\sigma_g \in \mathrm{GSub}(C)$ by construction of σ_g and $\sigma_g \in \mathrm{Sol}(C)$ by stability of $\succ$. Hence, $\sigma_g \in \mathrm{GSol}(C)$. In the following, we therefore concentrate on $\mathrm{Sol}(C)$ to determine the satisfiability of C.

To characterize some ground instances of a syntactic expression E (such as an equation or clause) by a constraint C expressing ordering and equality relations, we consider all substitutions $\sigma \in \mathrm{GSub}(E) \cap \mathrm{Sol}(C)$, which implies $\mathrm{Var}(E) \subseteq \mathrm{dom}(\sigma)$. Consider for example the constraint $f(x) > y \wedge y = f(z)$ to restrict the ground instances of $g(x, y, z) = x'$. It captures all ground substitutions σ with $\sigma(f(x)) \succ \sigma(y) \equiv \sigma(f(z))$, whereas $\sigma(x')$ is unconstrained (i. e., some arbitrary ground term). Note that this notion of satisfiability uses *extended signature semantics* (i. e., the ground substitutions are mappings into $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$) which is appropriate for deductive theorem proving.

The next examples show the versatility of ordering constraints.

EXAMPLE 5.1 *Consider the definition of critical pairs (see p. 12). Let $u = v$ and $s = t$ be two equations for which u unifies with $s|_p$. Let $\sigma = \mathrm{mgu}(u, s|_p)$. Then equation $\sigma(s)[\sigma(v)]_p = \sigma(t)$ is only considered as a critical pair if $\sigma(v) \not\succeq \sigma(u)$ and $\sigma(t) \not\succeq \sigma(s)$.*

With ordering constraints it is possible to formulate a condition that is more precise, namely, that at least one peak on the ground level has to exist. This is exactly the case when the constraint $\sigma(u) > \sigma(v) \wedge \sigma(s) > \sigma(t)$ is satisfiable. With the use of an equality constraint it is even possible to avoid the explicit computation of the unifier σ; the constraint is then $s|_p = u \wedge u > v \wedge s > t$. The ordering conditions in the inference are only a conservative approximation of the constraint-based formulation. Thus, unnecessary critical pairs may be generated, which can be avoided by the use of ordering constraints.

EXAMPLE 5.2 *With constraints it is possible to extend the reduction ordering. Instead of $\succ$ we can use $\succ_{\mathrm{cons}}$ which is defined as $s \succ_{\mathrm{cons}} t$ iff $t \geq s$ is unsatisfiable [KS95]. Whereas s and t might be incomparable in $\succ$, $\succ_{\mathrm{cons}}$ declares s to be greater than t if there is no ground instance such that the instance of t is greater or equal than the instance of s. In case of LPO, $\succ_{\mathrm{cons}}$ subsumes known extensions such as RDOS or KNSS (see e. g. [Ste94] for their definitions).*

As already evident from these small examples, ordering constraints are a powerful tool for the description of properties of the ground level on the term level. However, one should not underestimate the associated computational costs.

PROPOSITION 5.1 *Let $\succ$ be a ground reduction ordering. Determining the satisfiability of an ordering constraint is NP-hard in the general case.*

PROOF By reduction of 3-SAT. Let u, v, w be some ground terms with $u \succ v \succ w$. Replace in the 3-SAT-input each positive literal X_i by $x_i > v$ and each negative literal $\bar{X}_j$ by $v > x_j$. The resulting ordering constraint is satisfiable iff there is an assignment of the Boolean variables that satisfies the 3-SAT-input: If there is such an assignment, we can construct a satisfying substitution σ by $\sigma(x_i) = u$ if $X_i = 1$ and $\sigma(x_i) = w$ otherwise. Conversely, if σ satisfies the resulting ordering constraint, we can extract an assignment for the Boolean variables by $X_i = 1$ iff $\sigma(x_i) \succ v$. □

Note that in the proof we use an ordering constraint that is a conjunction containing disjunctions. In applications of ordering constraints, we often can restrict the form of

constraints to be only conjunctions of atomic constraints, which might be easier to handle. However, for LPO it was shown that the problem is NP-hard if the input consists of one single inequation [CT94].

For our applications, it is sufficient to consider quantifier-free constraints only. Variables are treated by Definition 5.2 as implicitly existentially quantified. Some researchers have investigated arbitrarily quantified constraints. Testing for satisfiability is then equivalent to deciding the full first-order theory of the reduction ordering $\succ$. For example, to describe simplification steps in [KKR90] the fragment $\forall^*\exists^*. C$ is used. This turned out to be undecidable in case $\succ$ is some LPO [CT97]. In [KV02] the first-order theory is shown to be decidable for KBO when $\mathcal{F}$ contains only constants and unary function symbols. In the recent [ZSM05] it is claimed that the full first-order theory of an arbitrary KBO is decidable.

5.2 A decision procedure for the satisfiability of LPO constraints

For $\succ$ being some LPO based on a total precedence $>_{\mathcal{F}}$, Comon showed the problem of determining the satisfiability of ordering constraints to be decidable [Com90]. Nieuwenhuis gave an NP-algorithm in [Nie93b]. In our implementation, we use the algorithm devised by Nieuwenhuis and Rivero [NR02] which is the fastest known method today. As our focus is on the efficient implementation of the method, the following presentation will deviate from the original one. There will be more emphasis on design decisions than on theoretical results. For the sake of clarity, we first present an overview of the algorithm introducing all concepts, sometimes in a simplified way. Then we discuss several refinements and optimizations.

An overview of the algorithm

The main idea is to successively decompose the constraints according to the definition of LPO until *solved forms* are reached for which it is easy to decide whether a solution exists. The structure of the solved forms has a profound influence on the whole algorithm. As they are essentially a conjunction of atomic constraints we keep the whole constraint in *disjunctive normal form.* This is the first design decision.

The *decomposition* of atomic constraints follows the different cases in the definition of LPO. For example the constraint $f(s_1, \ldots, s_n) > g(t_1, \ldots, t_m)$ is replaced by the disjunction $s_1 \geq g(t_1, \ldots, t_m) \vee \ldots \vee s_n \geq g(t_1, \ldots, t_m)$ if $f \not\geq_{\mathcal{F}} g$. This corresponds to the case (α) of LPO (cf. p. 33). It is easy to see that this transformation preserves the set of solutions. Because we keep the whole constraint in disjunctive normal form, such an introduction of additional disjunctions may lead to duplication of other parts of the constraint and thus to an excessive memory requirement. This can be seen if we enrich the example by some context. Replacing $f(s_1, \ldots, s_n) > g(t_1, \ldots, t_m) \wedge C \vee D$ by $s_1 \geq g(t_1, \ldots, t_m) \wedge C \vee \ldots \vee s_n \geq g(t_1, \ldots, t_m) \wedge C \vee D$ leads to n copies of C instead of one. The second design decision is therefore to represent disjunctions *implicitly* in an *or-tree* and to use *backtrack-search.* If the original constraint is not

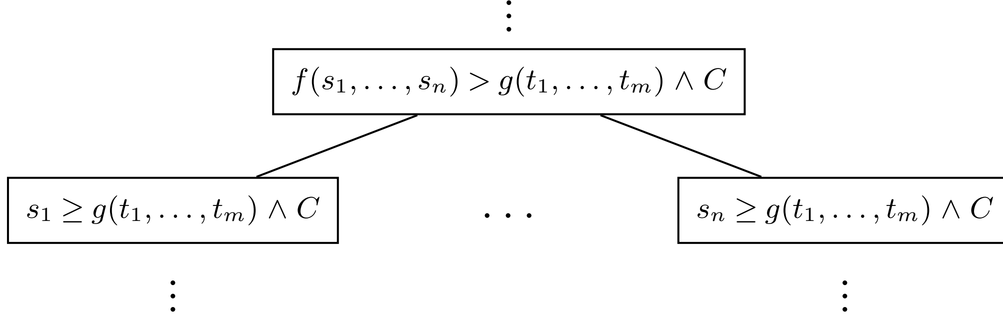


Figure 5.1: A fragment of the search tree corresponding to the decomposition of $f(s_1,\dots,s_n) > g(t_1,\dots,t_m)$ with $f \not\geq_{\mathcal{F}} g$.

a single conjunction, we transform it into disjunctive normal form and perform the backtrack-search for each conjunction one after the other.

Figure 5.1 depicts a fragment of the search tree corresponding to our example. As the different branches are developed successively, the duplication between the disjunctions is avoided. The overall memory requirement is therefore determined by the largest branch, not the whole tree. Furthermore, it is possible to share data structures along a branch. In the example, C is shared between the parent node and each of its child nodes. It is even possible to share the terms and subterms between the decomposed constraint and the newly constructed ones. The order in which the subtrees of a decomposition are expanded and the selection of the atom in the conjunction for decomposition are two important heuristic parameters of the algorithm. We assume that functions h_{Ord} and h_{Sel} provide this functionality.

Decomposition of atomic constraints takes place only if neither side of the constraint is a variable. The decomposition rules are depicted in Figure 5.2. Note the close relationship to the definition of LPO, e. g. dec_μ corresponds to the function majo in Section 4.2. In favor of clarity we use $\overline{s_n}$ as short-hand notation for $s_1,\dots,s_n$ and have furthermore omitted the conjunctive context C of the atom that is decomposed. Decomposition preserves the set of solutions, which can be shown easily by case analysis:

Lemma 5.1 *Let $C_1 \vee \dots \vee C_k$ result from C by decomposition of an atom in C with function* dec. *Then* $\mathrm{Sol}(C) = \bigcup_{i=1}^{k} \mathrm{Sol}(C_i)$.

Proof As the decomposition of $s \geq t$ and $s = t$ is easy we focus on the decomposition of $s > t$. Let $\sigma \in \mathrm{Sol}(s > t)$. Then $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t)$. Both terms are headed by a function symbol, hence, this must hold by cases (α), (β), or (γ) of the definition of LPO. Assume by case (α). Then $s \equiv f(s_1,\dots,s_n)$ and there is some $i \in \{1,\dots,n\}$ such that $\sigma(s_i) \succeq_{\mathrm{lpo}} \sigma(t)$. Therefore, $\sigma \in \mathrm{Sol}(s_i \geq t)$. Assume by case ($\beta$). Then $s \equiv f(s_1,\dots,s_n)$, $t \equiv g(t_1,\dots,t_m)$, $f >_{\mathcal{F}} g$, and $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t_k)$ for all $k \in \{1,\dots,m\}$. Therefore, $\sigma \in \mathrm{Sol}(s{>}t_1 \wedge \dots \wedge s{>}t_m)$, the constraint delivered by $\mathrm{dec}_\beta(s,t)$. Assume by case (γ). Then $s \equiv f(s_1,\dots,s_n)$, $t \equiv f(t_1,\dots,t_n)$, and there is some $i \in \{1,\dots,n\}$ such that $\sigma(s_j) \equiv \sigma(t_j)$ for all $j < i$, $\sigma(s_i) \succ_{\mathrm{lpo}} \sigma(t_i)$, and $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t_k)$ for all $k \in \{1,\dots,n\}$.

$$
\begin{aligned}
\mathrm{dec}(s > t) &= \mathrm{dec}_\alpha(\arg(s), t) \vee \mathrm{dec}_\beta(s,t) \vee \mathrm{dec}_\gamma(s,t) \\
\mathrm{dec}(s \geq t) &= s > t \vee s = t \\
\mathrm{dec}(s = t) &= \begin{cases} \mathrm{dec}_=(\arg(s), \arg(t)) & \text{if } \mathrm{top}(s) = \mathrm{top}(t) \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{dec}_=(\overline{s_n}, \overline{t_n}) &= s_1 = t_1 \wedge \ldots \wedge s_n = t_n \\
\mathrm{dec}_\alpha(\overline{s_n}, t) &= s_1 \geq t \vee \ldots \vee s_n \geq t \\
\mathrm{dec}_\beta(s,t) &= \begin{cases} \mathrm{dec}_\mu(s, \arg(t)) & \text{if } \mathrm{top}(s) >_{\mathcal{F}} \mathrm{top}(t) \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{dec}_\gamma(s,t) &= \begin{cases} \mathrm{dec}_\lambda(s, \arg(s), \arg(t)) & \text{if } \mathrm{top}(s) = \mathrm{top}(t) \\ \bot & \text{otherwise} \end{cases} \\
\mathrm{dec}_\mu(s, \overline{t_m}) &= s > t_1 \wedge \ldots \wedge s > t_m \\
\mathrm{dec}_\lambda(s, \overline{s_n}, \overline{t_n}) &= s_1 > t_1 \wedge \mathrm{dec}_\mu(s, \overline{t_n}) \vee \\
&\quad\; s_1 = t_1 \wedge s_2 > t_2 \wedge \mathrm{dec}_\mu(s, \overline{t_n}) \vee \ldots \vee \\
&\quad\; s_1 = t_1 \wedge \ldots \wedge s_{n-1} = t_{n-1} \wedge s_n > t_n \wedge \mathrm{dec}_\mu(s, \overline{t_n})
\end{aligned}
$$

Figure 5.2: Decomposition rules for atomic constraints with nonvariable sides.

Therefore, $\sigma \in \mathrm{Sol}(s_1 = t_1 \wedge \ldots \wedge s_{i-1} = t_{i-1} \wedge s_i > t_i \wedge s > t_1 \wedge \ldots \wedge s > t_n)$, the ith constraint delivered by $\mathrm{dec}_\gamma(s,t)$. Conversely, let $\sigma \in \mathrm{Sol}(C_i)$ for constraint C_i delivered by dec_α, dec_β, or dec_γ. Then, $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t)$ by case (α), case (β), or case (γ) of LPO. Hence, $\sigma \in \mathrm{Sol}(s > t)$. □

Analogously to unification, we propagate the knowledge contained in an equality constraint with a variable in one of its sides into the rest of the conjunction.

$$
\mathrm{prop}(x{=}t \wedge C_1) = \begin{cases} x = t \wedge \theta(C_1) & \text{if } x \notin \mathrm{Var}(t) \text{ and } x \in \mathrm{Var}(C_1), \text{ where } \theta = \{x \mapsto t\} \\ \bot & \text{if } x \in \mathrm{Var}(t) \\ x = t \wedge C_1 & \text{otherwise} \end{cases}
$$

Lemma 5.2 *Let C' result from C by propagation of a variable binding in C. Then* $\mathrm{Sol}(C) = \mathrm{Sol}(C')$.

Proof If $C' \equiv \bot$ then C contains an equality constraint $x = t$ with $x \in \mathrm{Var}(t)$. This implies $\mathrm{Sol}(x = t) = \varnothing$ and $\mathrm{Sol}(C) = \varnothing$. If C' is different from C then C contains an equality constraint $x = t$ with $x \notin \mathrm{Var}(t)$ and $x \in \mathrm{Var}(C_1)$. Let σ be a substitution with $\sigma(x) \equiv \sigma(t)$ (i. e., $\sigma \in \mathrm{Sol}(x = t)$). Then $\sigma(s) \equiv \sigma(\theta(s))$ for any term s. Hence, $\sigma(C) \equiv \sigma(C')$. Therefore, $\mathrm{Sol}(C) = \mathrm{Sol}(C')$, as $\mathrm{Sol}(C) \subseteq \mathrm{Sol}(x = t)$ and $\mathrm{Sol}(C') \subseteq \mathrm{Sol}(x = t)$. □

The propagation of a variable binding requires another design decision: Should we perform substitutions explicitly or should we keep them implicit and use an explicit *binding context*? Here, the underlying term data structure is important. Applying a substitution to a flat-term is relatively expensive, because the new term has to be built from fresh term cells. Considering the backtracking search, it seems more appropriate to use a binding context, for which it is cheap to add or to remove additional bindings. These operations are simple pointer assignments. The original binding has to be kept in any case. Hence the subterms of the original constraint can be completely shared during the backtrack-search. The same benefit is provided by Prolog-terms (see e. g. [RSV01]) which are especially designed for such situations. Nevertheless, we think that it is easier to make the functions called by the constraint solver aware of the binding context than to introduce a completely new term data structure into the system. To simplify the presentation we will mention the binding context only when it is needed and we will assume that all used functions will respect it.

With this design decision we have split the search state in each node of the tree in a *solved part* S (containing at least the binding context B) and an *unsolved part* U. Such a distinction is common in similar algorithms, e. g. for unification. During the search, we deplete the unsolved part and enrich the solved part until the unsolved part is empty and we have reached a solved form. But before we can define what a solved form is, we have to introduce two other concepts, namely *consequences by transitivity* and a notion of *redundancy*.

The first one realizes a very restricted form of transitivity. This is necessary because any constraint $s > t$ can equally be expressed as $s \geq x \wedge x > t$, with x a new variable. Whereas for the first form decompositions may take place, for the latter form they may not, as both atomic constraints have a variable at one of their sides.

DEFINITION 5.3 *For two atomic constraints* $s \varrho_1 x$ *and* $x \varrho_2 t$ *with* $s \notin \mathcal{V}$, $x \in \mathcal{V}$, *and* $\varrho_1, \varrho_2 \in \{>, \geq\}$ *the atom* $s \varrho t$ *with* $\varrho = \max_{\sqsupset}\{\varrho_1, \varrho_2\}$ *is a* consequence by transitivity.

Note that for atoms where neither side is a variable no consequences need to be inferred as such atoms are subject to decomposition. In [NR02] it is shown that consequences with $s \in \mathcal{V}$ are also unnecessary; only the cases covered by Definition 5.3 are needed for the proofs.

EXAMPLE 5.3 *The constraint* $f(x) > z$ *is a consequence by transitivity of* $f(x) \geq y$ *and* $y > z$. *From* $x \geq y$ *and* $y > z$ *there is no consequence by transitivity, nor for* $f(x) \geq a$ *and* $a > z$, *nor for* $f(x) \geq y$ *and* $y = z$.

LEMMA 5.3 *Let constraint* $s \varrho t$ *be a consequence by transitivity of* $s \varrho_1 x$ *and* $x \varrho_2 t$. *Then* $\mathrm{Sol}(s \varrho_1 x \wedge x \varrho_2 t) \subseteq \mathrm{Sol}(s \varrho t)$.

PROOF Because $\succcurlyeq_{\mathrm{lpo}}$ is transitive any solution of $s \varrho_1 x \wedge x \varrho_2 t$ satisfies $s \varrho t$. □

The addition of consequences by transitivity to a conjunction therefore does not change the set of solutions.

COROLLARY 5.1 *Let $s \varrho t$ be a consequence by transitivity of two atomic constraints contained in the conjunction C. Then* $\mathrm{Sol}(C) = \mathrm{Sol}(s \varrho t \wedge C)$. □

The second concept is an adequate notion of redundancy. If we have for $s>t \wedge C$ that $\mathrm{Sol}(s > t) \supseteq \mathrm{Sol}(C)$ then the atomic constraint gives no additional information about the set of solutions, it is *redundant* with respect to C. This general notion of redundancy is only of theoretical interest as it entails the original problem of the satisfiability of a constraint. A practical, and for our purposes sufficient, approximation is provided by the following definition. It extends the usual definitions of syntactic equality and of LPO by some information contained in C.

DEFINITION 5.4 *Let $>_{\mathcal{F}}$ be a total precedence, $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$, and C a conjunction of atomic constraints.*

The syntactic equality extended by C *is given as $s \equiv^C t$ iff either $s \equiv t$, or $s = t$ is in C, or $s \equiv f(s_1, \ldots, s_n)$, $t \equiv f(t_1, \ldots, t_n)$ and $s_i \equiv^C t_i$ for all $i \in \{1, \ldots, n\}$.*

The LPO extended by C *is given as $s \succ_{\mathrm{lpo}}^C t$ iff either $s \equiv f(s_1, \ldots, s_n)$, $t \equiv g(t_1, \ldots, t_m)$, and*

(α) *$s_i \succcurlyeq_{\mathrm{lpo}}^C t$ for some $i \in \{1, \ldots, n\}$ or*

(β) *$f >_{\mathcal{F}} g$ and $s \succ_{\mathrm{lpo}}^C t_k$ for all $k \in \{1, \ldots, m\}$ or*

(γ) *$f = g$, there exists some $i \in \{1, \ldots, m\}$ such that $s_j \equiv^C t_j$ for all $j \in \{1, \ldots, i-1\}$ and $s_i \succ_{\mathrm{lpo}}^C t_i$, and $s \succ_{\mathrm{lpo}}^C t_k$ for all $k \in \{1, \ldots, m\}$*

or $s \equiv f(s_1, \ldots, s_n)$, $t \equiv x$, $x \in \mathcal{V}$, and

(δ) *$s_i \succcurlyeq_{\mathrm{lpo}}^C x$ for some $i \in \{1, \ldots, n\}$,*

or

(ε) *$s > u$ is in C and $u \succcurlyeq_{\mathrm{st}} t$ or $s \varrho u$ is in C with $\varrho \in \{=, \geq\}$ and $u \succ_{\mathrm{st}} t$,*

where $s \succcurlyeq_{\mathrm{lpo}}^C t$ is given by $s \succcurlyeq_{\mathrm{lpo}}^C t$ iff $s \equiv^C t$, or $s \succ_{\mathrm{lpo}}^C t$, or

(ε') *$s \varrho u$ is in C with $\varrho \in \{=, >, \geq\}$ and $u \succcurlyeq_{\mathrm{st}} t$.*

An atom A is redundant with respect to C *if either A is $s = t$ and $s \equiv^C t$ holds, or A is $s > t$ and $s \succ_{\mathrm{lpo}}^C t$ holds, or A is $s \geq t$ and $s \succcurlyeq_{\mathrm{lpo}}^C t$ holds.*

Note that if C contains atom A, then A is redundant with respect to C. The definition of the LPO extended by C closely follows the definition of LPO (cf. p. 33) and both relations coincide if C is trivial (i. e., $C \equiv \top$).

EXAMPLE 5.4 *Let $a >_{\mathcal{F}} f >_{\mathcal{F}} g$ and let C contain the atoms $x > a$ and $y \geq g(z)$. Then $f(x, y) \succ_{\mathrm{lpo}}^C f(a, f(x, z))$, but $f(x, y) \not\succ_{\mathrm{lpo}} f(a, f(x, z))$. If C contains additionally $a > f(x, y)$ we have $f(x, y) \succ_{\mathrm{lpo}}^C f(a, f(x, z)) \succ_{\mathrm{lpo}}^C f(x, y)$, but $f(x, y) \not\succ_{\mathrm{lpo}}^C f(x, y)$. Hence, for unsatisfiable C, relation $\succ_{\mathrm{lpo}}^C$ is not necessarily an ordering.*

For solutions of C, the relations extended by C describe the equality and ordering relations on the instances.

LEMMA 5.4 *Let $\sigma \in \mathrm{Sol}(C)$. If $s \equiv^C t$ then $\sigma(s) \equiv \sigma(t)$, if $s \succ_{\mathrm{lpo}}^C t$ then $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t)$, and if $s \succcurlyeq_{\mathrm{lpo}}^C t$ then $\sigma(s) \succcurlyeq_{\mathrm{lpo}} \sigma(t)$.*

PROOF Induction on $|s| + |t|$. Consider $s \equiv^C t$. Either $s \equiv t$ which trivially implies $\sigma(s) \equiv \sigma(t)$, or $s = t$ for which $\sigma \in \mathrm{Sol}(s = t)$ implies $\sigma(s) \equiv \sigma(t)$, or $s \equiv f(s_1, \ldots, s_n)$, $t \equiv f(t_1, \ldots, t_n)$ and $s_i \equiv^C t_i$ for all $i \in \{1, \ldots, n\}$, which implies by induction hypothesis $\sigma(s_i) \equiv \sigma(t_i)$ for all $i \in \{1, \ldots, n\}$, hence $\sigma(s) \equiv \sigma(t)$.

Consider $s \succ^C_{\mathrm{lpo}} t$. Because cases (α) to (δ) follow closely their counterparts in the definition of LPO they are easy consequences of induction hypothesis. Case (ε): If $s \succ^C_{\mathrm{lpo}} t$ because $s > u$ is in C and $u \succeq_{\mathrm{st}} t$ or $s \varrho u$ is in C with $\varrho \in \{=, \geq\}$ and $u \succ_{\mathrm{st}} t$, then for all $\sigma \in \mathrm{Sol}(C)$ it follows that $\sigma(s) \succ_{\mathrm{lpo}} \sigma(u) \succeq_{\mathrm{st}} \sigma(t)$ or $\sigma(s) \succeq_{\mathrm{lpo}} \sigma(u) \succ_{\mathrm{st}} \sigma(t)$. As $\succ_{\mathrm{st}} \subseteq \succ_{\mathrm{lpo}}$ both cases imply $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t)$.

Consider $s \succeq^C_{\mathrm{lpo}} t$. As already discussed, $s \equiv^C t$ and $s \succ^C_{\mathrm{lpo}} t$ imply $\sigma(s) \equiv \sigma(t)$ and $\sigma(s) \succ_{\mathrm{lpo}} \sigma(t)$, hence $\sigma(s) \succeq_{\mathrm{lpo}} \sigma(t)$ holds in these cases. Case (ε'): If $s \succeq^C_{\mathrm{lpo}} t$ because $s \varrho u$ is in C with $\varrho \in \{=, >, \geq\}$ and $u \succeq_{\mathrm{st}} t$ then for all $\sigma \in \mathrm{Sol}(C)$ it follows that $\sigma(s) \succeq_{\mathrm{lpo}} \sigma(u) \succeq_{\mathrm{st}} \sigma(t)$. As the LPO contains the subterm relation and is transitive, $\sigma(s) \succeq_{\mathrm{lpo}} \sigma(t)$ follows. □

Therefore, we can delete redundant atoms without changing the set of solutions.

COROLLARY 5.2 *If atom A is redundant with respect to C, then* $\mathrm{Sol}(C) \subseteq \mathrm{Sol}(A)$ *and therefore* $\mathrm{Sol}(A \wedge C) = \mathrm{Sol}(C)$. □

Furthermore, the LPO extended by C gives a sufficient test for unsatisfiability.

COROLLARY 5.3 *If* $t \succeq^C_{\mathrm{lpo}} s$ *then* $\mathrm{Sol}(s > t \wedge C) = \varnothing$. *If* $t \succ^C_{\mathrm{lpo}} s$ *then* $\mathrm{Sol}(s \geq t \wedge C) = \varnothing$. *If* $s \succ^C_{\mathrm{lpo}} t$ *or* $t \succ^C_{\mathrm{lpo}} s$ *then* $\mathrm{Sol}(s = t \wedge C) = \varnothing$. □

This allows us to use the LPO extended by C to prune the search tree.

The implementation of $\equiv^C$ is straightforward. For the implementation of $\succ^C_{\mathrm{lpo}}$ we can extend the implementation of $\succ_{\mathrm{lpo}}$ as developed in Section 4.2 to handle additionally the cases (ε) and (ε'). Furthermore, we have to adapt the case (δ) by replacing the optimized variable test with a call to the code for case (α). All the optimizations presented during the development of an efficient version of $\succ_{\mathrm{lpo}}$ concern cases (α) to (γ) and are therefore applicable without modification. For terms s and t and constraint C, cases (ε) and (ε') can be implemented in $O(|C| + |s| + |t|)$ worst case running time. Taking recursive calls via cases (α) to (δ) into account, this gives an $O(|C| \cdot |s| \cdot |t|)$ worst case running time for $s \succ^C_{\mathrm{lpo}} t$ and $s \succeq^C_{\mathrm{lpo}} t$. This means that we can get polynomial behavior without the use of dynamic programming techniques. In [NR02] it is suggested to use dynamic programming to achieve a polynomial time algorithm.

Note that this notion of redundancy is stable against the application of substitutions, the deletion of redundant atoms, and the decomposition process. That is, if atom A is redundant with respect to C, then $\sigma(A)$ is redundant with respect to $\sigma(C)$. If A is redundant with respect to $A' \wedge C$ and A' is redundant with respect to C, then A is redundant with respect to C. If C' is derived from C by decomposition of atom A then the redundancy of A' with respect to C implies the redundancy of A' with respect to C'. Furthermore, the decomposed atom A is itself redundant with respect to C'. These properties can be shown by induction on the structure of terms combined with an extensive case analysis.

We are now able to define the concept of a solved form.

Definition 5.5 *A conjunction of atomic inequality constraints C is a* solved form *if*

(1) no atom A contained in C is redundant with respect to C without A,
(2) no atom A contained in C can be further decomposed, and
(3) any consequence by transitivity A of two atoms A_1 and A_2 contained in C is redundant with respect to C.

A conjunction S is a solved form of a constraint *C if it is a solved form and derived from C with decomposition and propagation of variable bindings, deletion of redundant atoms, and addition of nonredundant consequences by transitivity.*

As mentioned before, the search state C in a node consists of an unsolved part U and a solved part S with $C \equiv S \wedge U$. The binding context B is part of S. An invariant of the algorithm is that S always fulfills conditions (1) and (2). The next important design decision is that only S induces the redundancy relation and consequences by transitivity are inferred only from atoms in S. To establish condition (1), redundant atoms in S have to be eliminated, that is, no atom A in S is redundant with respect to S without A. Because of condition (2), S contains only atoms that cannot be further decomposed (i. e., where at least one side is a variable). This simplifies the implementation of the relations needed to test for redundancy.

At each iteration of the search one atom A in U is selected by some heuristics h_{Sel}. If it is redundant we delete it. If we can show A unsatisfiable with the help of $\succ^S_{\mathrm{lpo}}$ we abandon this branch of the search tree and backtrack to the last branching decomposition node. If A is subject to decomposition, for each conjunction C_i returned by $\mathrm{dec}(A)$ a new branch is started with A replaced by C_i. The order of the subtrees is determined by the heuristics h_{Ord}. If A is not subject to decomposition, it is transferred to S. As a result some atoms in S may become redundant and are deleted. If A is an equality constraint $x = t$, the binding context is updated accordingly. Furthermore, atoms $s \varrho x$ or $x \varrho u$ in S are deleted and atoms $s \varrho t$ or $t \varrho u$ are added to U.

To achieve condition (3), that is, to make all consequences by transitivity of atoms in S redundant, we compute all of them and add the nonredundant ones to U (provided they do not already occur in U). We adopt an eager strategy for adding transitive consequences: As soon as both “parents” are in S we add the consequence. An alternative is to use a lazy approach and delay the generation of transitive consequences: When U is empty one transitive consequence of atoms in S is inferred and put into U. For the latter strategy an explicit administration component is necessary to ensure that each consequence by transitivity is considered once. Alternatively, to avoid the administration overhead, one can compute all transitive consequences of S when U is empty and check eagerly for redundancy. Of course, this might lead to transitive consequences that are computed several times. In [NR02] a lazy approach is suggested. However, in our experience, there are no advantages of delaying the generation of consequences by transitivity. Hence, we have chosen the eager approach for its simplicity.

When U becomes empty and all consequences by transitivity are inferred, S fulfills all three conditions of Definition 5.5. Therefore, we have found a solved form.

Lemma 5.5 *The derivation process terminates, that is, the search tree has no infinite branch.*

Proof We first define the following notions of complexities of $S \wedge U$: Let N_1 be the number of variables in $S \wedge U$ that are not bound in B. Let T be the set of all subterms in $S \wedge U$ and $T_2 = T \times T$. Let T_2' be defined as $T_2' = \{(s,t) \mid (s,t) \in T_2, s \succcurlyeq_{\text{lpo}}^{S \wedge U} t$ or $t \succcurlyeq_{\text{lpo}}^{S \wedge U} s\}$ and let $N_2 = |T_2 - T_2'|$. The complexity of an atom $s \varrho t$ is the multiset $\{|s|, |t|, i\}$ where $i = 1$ if $\varrho = \geq$ and $i = 0$ otherwise. Finally, let M be the multiset of complexities of the atoms in $S \wedge U$.

The complexity of $S \wedge U$ is then the tuple (N_1, N_2, M). We show that with each step in the derivation process this complexity decreases with respect to ordering $> = (>_{\mathbb{N}}, >_{\mathbb{N}}, \ggg_{\mathbb{N}})$ where $\ggg_{\mathbb{N}}$ is the multiset extension of the multiset extension of $>_{\mathbb{N}}$. Note that by construction $>$ is a well-founded ordering.

The propagation of variables decreases N_1. The addition of nonredundant consequences by transitivity leaves N_1 unaffected and decreases N_2. The deletion of redundant atoms possibly decreases N_1 or N_2, and definitely decreases M. The decomposition of atoms does not affect N_1, may decrease N_2, and definitely decreases M. □

Lemma 5.6 *For each constraint C we can compute a finite number $S_1, \ldots, S_k$ of solved forms for C with* $\mathrm{Sol}(C) = \bigcup_{i=1}^{k} \mathrm{Sol}(S_i)$.

Proof The solved forms for C are at the ends of the branches of the search tree for C. (There might be abandoned branches with an unsatisfiable node at the end.) By the previous lemma, each branch in the tree is finite and the tree is finitely branching – only the decomposition of inequalities introduces new branches. So it has only a finite number of leaves. Lemmas 5.1 and 5.2 and Corollaries 5.1 and 5.2 ensure that the set of solutions is preserved. □

To decide the satisfiability of a solved form we need the following concept of *cycles.*

Definition 5.6 *A conjunction of atoms C has a* cycle *if there is some n and*

$$x_1 \,\varrho_1\, t_1[x_2]_{p_1}, \quad x_2 \,\varrho_2\, t_2[x_3]_{p_2}, \quad \ldots, \quad x_n \,\varrho_n\, t_n[x_1]_{p_n}$$

are in C where $x_i \in \mathcal{V}$ and $\varrho_i \in \{>, \geq\}$ for all $i \in \{1, \ldots, n\}$, and for at least one $j \in \{1, \ldots, n\}$ either $p_j \neq \lambda$ or $\varrho_j = >$.

Example 5.5 *Let $C \equiv x \geq f(a,y) \wedge y \geq z \wedge z > x \wedge y \geq g(x) \wedge z \geq y \wedge g(a) > x$. It has two cycles. One is established by the first three atoms. The other results from the first and the forth atom. Atoms like the last one do not contribute to cycles because they have a nonvariable left-hand side. Atoms $y \geq z$ and $z \geq y$ do not generate a cycle either, because both constraint symbols are $\geq$ and the variables at the right-hand sides occur top-level.*

It is clear that a conjunction with a cycle is unsatisfiable.

Lemma 5.7 *Let C be a conjunction of atoms with a cycle. Then* $\mathrm{Sol}(C) = \varnothing$.

Proof As LPO is transitive and contains the subterm relation a cycle in C implies that $\sigma(x_1) \succ_{\mathrm{lpo}} \sigma(x_1)$ for all $\sigma \in \mathrm{Sol}(C)$, which violates that LPO is irreflexive. □

However, a solved form is satisfiable if it has no cycle (for the proof we refer to [NR02]).

Theorem 5.1 *Let S be a solved form. Then S is satisfiable iff S has no cycle.* □

Corollary 5.4 *A constraint C is satisfiable if at least one of its solved forms has no cycle.* □

For deciding the satisfiability of an LPO constraint C it therefore suffices to test the solved forms of C for cycles. This is of course interleaved in the backtrack search. As soon as one solved form is cycle-free, we can stop the search and return "satisfiable". Conversely, cycle detection is a further means to prune the search tree when applied to $S \wedge U$ before U is empty. If all branches of the search tree lead to unsatisfiable constraints, then C is unsatisfiable as well and "unsatisfiable" is returned.

To implement a test for cycles in S we derive a quasi ordering $\succsim_v$ on $\mathcal{V}$ from the inequality atoms in S that have a variable as left-hand side. From an atom $x\,\varrho\,t[y]_p$, $\varrho \in \{>, \geq\}$, we derive $x \succ_v y$ if $\varrho = >$ or $p \neq \lambda$. Otherwise, it is of the form $x \geq y$ and we derive $x \succsim_v y$. Then we compute the transitive closure. If we infer $x \succ_v x$ for some variable x then there is a cycle in S. The problem is that $\succsim_v$ is a quasi-ordering. Hence, we have to take into account the equivalence relation $\approx_v$.

Improvements and refinements

For the first improvement consider the definition of the decomposition rules of Figure 5.2 which follows very closely the definition of LPO. The results of Section 4.2 suggest that this is rather naive and can be optimized. This is indeed the case. Figure 5.3 contains the optimized variants of the decomposition rules. We can distinguish two kinds of optimizations. The first avoids the extension of conjunctions by atoms for which we know a priori that they are redundant. For example, consider a conjunction introduced by dec_λ. Function dec_μ adds n atoms of the form $s > t_i$. However, atom $s > t_i$ is redundant if $s_i = t_i$ or $s_i > t_i$ is present. Hence, such atoms can be omitted. In consequence, all the conjunctions delivered by dec'_λ consist of n atoms.

The second kind of optimizations is more important. It avoids the generation of some of the conjunctions. Therefore, the branching factor of the rules is considerably reduced. For example, let $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv g(t_1, \ldots, t_m)$. If $f >_{\mathcal{F}} g$ then the constraints corresponding to case (α) can be omitted completely. In case of $f = g$, the first conjunction delivered by dec_α is covered by the first lexicographic case. This can be generalized, $s_i \geq t$ is covered by the ith lexicographic case if $s_j \equiv t_j$ for all $j < i$. This justifies the restrictions in dec'_α. For decomposing an atom $s \geq t$ it is not necessary to generate $s > t \vee s = t$. The equality constraint is only satisfiable if both terms have the same top-symbols. If this is the case, we get $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$. Hence, this case can be combined with the last conjunction generated for the lexicographic comparison

$$\mathrm{dec}'(s > t) = \begin{cases} \mathrm{dec}_\alpha(\mathrm{arg}(s), t) & \text{if } \mathrm{top}(s) \not\geq_{\mathcal{F}} \mathrm{top}(t) \\ \mathrm{dec}_\mu(s, \mathrm{arg}(t)) & \text{if } \mathrm{top}(s) >_{\mathcal{F}} \mathrm{top}(t) \\ \mathrm{dec}'_\lambda(s, \mathrm{arg}(s), \mathrm{arg}(t), >) \vee \mathrm{dec}'_\alpha(\mathrm{arg}(s), t, \mathrm{arg}(t)) & \text{if } \mathrm{top}(s) = \mathrm{top}(t) \end{cases}$$

$$\mathrm{dec}'(s \geq t) = \begin{cases} \mathrm{dec}_\alpha(\mathrm{arg}(s), t) & \text{if } \mathrm{top}(s) \not\geq_{\mathcal{F}} \mathrm{top}(t) \\ \mathrm{dec}_\mu(s, \mathrm{arg}(t)) & \text{if } \mathrm{top}(s) >_{\mathcal{F}} \mathrm{top}(t) \\ \mathrm{dec}'_\lambda(s, \mathrm{arg}(s), \mathrm{arg}(t), \geq) \vee \mathrm{dec}'_\alpha(\mathrm{arg}(s), t, \mathrm{arg}(t)) & \text{if } \mathrm{top}(s) = \mathrm{top}(t) \end{cases}$$

$$\mathrm{dec}'(s = t) = \mathrm{dec}(s = t)$$

$$\begin{aligned} \mathrm{dec}'_\lambda(s, \overline{s_n}, \overline{t_n}, \varrho) = {} & s_1 > t_1 \wedge s > t_2 \wedge \ldots \wedge s > t_n \vee \\ & s_1 = t_1 \wedge s_2 > t_2 \wedge s > t_3 \wedge \ldots \wedge s > t_n \vee \ldots \vee \\ & s_1 = t_1 \wedge \ldots \wedge s_{n-2} = t_{n-2} \wedge s_{n-1} > t_{n-1} \wedge s > t_n \vee \\ & s_1 = t_1 \wedge \ldots \wedge s_{n-1} = t_{n-1} \wedge s_n \,\varrho\, t_n \\ & \text{where } \varrho \in \{>, \geq\} \text{ must hold} \end{aligned}$$

$$\mathrm{dec}'_\alpha(\overline{s_n}, t, \overline{t_n}) = \begin{cases} s_{i+1} > t \vee \ldots \vee s_n > t & \text{if } s_j \equiv t_j \text{ for all } j < i \text{ and } s_i \not\equiv t_i \\ \bot & \text{if } s_j \equiv t_j \text{ for all } j \leq n \end{cases}$$

Figure 5.3: Improved decomposition rules for inequality constraints.

which is $s_1 = t_1 \wedge \ldots \wedge s_{n-1} = t_{n-1} \wedge s_n > t_n$. Therefore, the handling of $\geq$-constraints can be made nearly identical to the handling of $>$-constraints, they only differ in the last case of dec'_λ.

LEMMA 5.8 *Let $C_1 \vee \ldots \vee C_k$ result from C by decomposition of an atom in C with function* dec'. *Then* $\mathrm{Sol}(C) = \bigcup_{i=1}^{k} \mathrm{Sol}(C_i)$.

PROOF Because of Lemma 5.1 we only have to justify the differences to decomposition by function dec. Let $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv g(t_1, \ldots, t_m)$. Consider $\mathrm{dec}'(s > t)$. If $\mathrm{top}(s) \not\geq_{\mathcal{F}} \mathrm{top}(t)$ then nothing is changed. If $\mathrm{top}(s) >_{\mathcal{F}} \mathrm{top}(t)$ then the atoms delivered by dec_α are omitted. Let $s_i \geq t$ be one of these atoms. Then $\mathrm{Sol}(s_i \geq t) \subseteq \mathrm{Sol}(\mathrm{dec}_\mu(s, \mathrm{arg}(t)))$ which is an easy consequence of Lemma 4.2. If $f = g$ then the conjunctions concerning the lexicographic case contain less atoms and the number of atoms concerning case (α) are reduced. The first optimization is justified by Lemma 4.1: Atom $s{>}t_i$ is redundant if $s_i{=}t_i$ or $s_i{>}t_i$ is present, because $\mathrm{Sol}(s_i{=}t_i) \subseteq \mathrm{Sol}(s{>}t_i)$ and $\mathrm{Sol}(s_i > t_i) \subseteq \mathrm{Sol}(s > t_i)$. For the second optimization it is clear that $\mathrm{Sol}(s_j \geq t) = \varnothing$ for all atoms with $s_j \equiv t_j$. Furthermore, $\mathrm{Sol}(s_i \geq t) \subseteq \mathrm{Sol}(s_1 = t_1 \wedge \ldots \wedge s_{i-1} = t_{i-1} \wedge s_i > t_i \wedge s > t_{i+1} \wedge \ldots \wedge s > t_n)$ if $s_j \equiv t_j$ for all $j < i$, which is also a consequence of Lemma 4.1. Finally, the improved decomposition of $s \geq t$ takes into

account that constraint $s = t$ requires identical top-symbols to be satisfiable. The then generated constraint $s_1 = t_1 \wedge \ldots \wedge s_{n-1} = t_{n-1} \wedge s_n = t_n$ is obviously covered by $s_1 = t_1 \wedge \ldots \wedge s_{n-1} = t_{n-1} \wedge s_n \geq t_n$, the last conjunction delivered by $\mathrm{dec}'(s \geq t)$ for the lexicographic comparison. □

A second topic concerns the notion of redundancy. In [NR02] it is suggested to extend the definition of $s \succcurlyeq^C_{\mathrm{lpo}} t$ if $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv f(t_1, \ldots, t_n)$ by the additional case

(γ') $s_i \succcurlyeq^C_{\mathrm{lpo}} t_i$ for all $i \in \{1, \ldots, n\}$.

This is in our experience a valuable extension. It strengthens not only $\succcurlyeq^C_{\mathrm{lpo}}$ but, via recursive calls, also $\succ^C_{\mathrm{lpo}}$.

The definition of consequences by transitivity is restrictive in that it requires the left-hand side of the resulting atom to be a nonvariable term. Other consequences are unnecessary for theoretical reasons (cf. [NR02]). However, we are free to add additional consequences as long as they do not change satisfiability. Termination is not affected if we add only nonredundant ones (cf. Lemma 5.5).

Definition 5.7 *For two atomic constraints* $x \,\varrho_1\, y$ *and* $y \,\varrho_2\, t$ *with* $x, y \in \mathcal{V}$*, and* $\varrho_1, \varrho_2 \in \{>, \geq\}$ *the atom* $x \,\varrho\, t$ *with* $\varrho = \max_{\sqsupset}\{\varrho_1, \varrho_2\}$ *is a* supplementary inequality constraint. *For* $s \geq t$ *and* $t \geq s$ *the atom* $s = t$ *is a* supplementary equality constraint. Supplementary consequences by transitivity *are supplementary inequality constraints or supplementary equality constraints.*

Lemma 5.9 *Let atom* A *be a supplementary consequence by transitivity of atoms* A_1 *and* A_2*. Then* $\mathrm{Sol}(A_1 \wedge A_2) \subseteq \mathrm{Sol}(A)$*.*

Proof If A is a supplementary inequality constraint then $A_1 \equiv x \,\varrho_1\, y$, $A_2 \equiv y \,\varrho_2\, t$, and $A \equiv x \,\varrho\, t$ with $\varrho = \max_{\sqsupset}\{\varrho_1, \varrho_2\}$. Therefore, any solution of $x \,\varrho_1\, y \wedge y \,\varrho_2\, t$ satisfies $x \,\varrho\, t$ as $\succcurlyeq_{\mathrm{lpo}}$ is transitive. If A is a supplementary equality constraint then $A_1 \equiv s \geq t$ and $A_2 \equiv t \geq s$. Any solution of A_1 and A_2 maps s and t to the same term, hence unifies s and t. □

By adding supplementary consequences by transitivity we can use a simpler form of cycle detection by restricting our attendance to *strict cycles.*

Definition 5.8 *A conjunction of atoms* C *has a* strict cycle *if there is some* n *and*

$$x_1 \,\varrho_1\, t_1[x_2]_{p_1}, \quad x_2 \,\varrho_2\, t_2[x_3]_{p_2}, \quad \ldots, \quad x_n \,\varrho_n\, t_n[x_1]_{p_n}$$

are in C *where* $x_i \in \mathcal{V}$ *and* $\varrho_i \in \{>, \geq\}$ *for all* $i \in \{1, \ldots, n\}$*, and for all* $j \in \{1, \ldots, n\}$ *either* $p_j \neq \lambda$ *or* $\varrho_j = >$*.*

Note that a cycle is not a strict cycle if it contains atoms of the form $x_i \geq x_j$. We call such atoms *nonstrict cycle elements.* Of course, any strict cycle is also a cycle.

EXAMPLE 5.6 *Let* $C \equiv x \geq f(a,y) \wedge y \geq z \wedge z > x$. *It has a cycle, but not a strict cycle. Let* C' *result from* C *by adding the supplementary transitive consequence* $y > x$. *Then* C' *contains a strict cycle.*

LEMMA 5.10 *Let* C *be a solved form and all supplementary transitive consequences of atoms in* C *are redundant with respect to* C. *Then* C *contains a cycle iff* C *contains a strict cycle.*

PROOF As any strict cycle is a cycle, we only have to consider the case when C contains a cycle, but no strict cycle. Let $x_1 \,\varrho_1\, t_1[x_2]_{p_1}, x_2 \,\varrho_2\, t_2[x_3]_{p_2}, \ldots, x_n \,\varrho_n\, t_n[x_1]_{p_n}$ be a cycle in C with a minimal number of nonstrict cycle elements. There must be at least one nonstrict cycle element, otherwise the cycle was also a strict cycle. Let $x_j \geq x_k$ be one of these. If $j = n$ then $k = 1$, otherwise $k = j + 1$. Because C contains a minimal number of nonstrict cycle elements the supplementary consequence by transitivity $x_j \,\varrho_k\, t_k$ must be redundant in C. Otherwise, we could replace $x_j \geq x_k$ and $x_k \,\varrho_k\, t_k$ by $x_j \,\varrho_k\, t_k$ and get a cycle with fewer nonstrict cycle elements. Therefore, there must be some atom $x_j \,\varrho'\, u$ in C with $u \succcurlyeq_{\mathrm{st}} t_k$ such that it makes $x_j \,\varrho_k\, t_k$ redundant. Because $\mathrm{Var}(t_k) \subseteq \mathrm{Var}(u)$, we can replace the two atoms $x_j \geq x_k$ and $x_k \,\varrho_k\, t_k$ by $x_j \,\varrho'\, u$. The result is still a cycle, but contains fewer nonstrict cycle elements. Hence there must be a strict cycle in C. □

The implementation of a test for strict cycles is similar to the test for normal cycles. However, it can be considerably simplified because we do not have to consider atoms of the form $x \geq y$. Hence, we derive a *partial* ordering $\succ_v$ on $\mathcal{V}$ from the inequality atoms that have a variable as left-hand side. Let C be a solved form and and all supplementary transitive consequences of atoms in C are redundant with respect to C. From an atom $x \,\varrho\, t[y]_p$, $\varrho \in \{>, \geq\}$, we derive $x \succ_v y$ if $\varrho = >$ or $p \neq \lambda$. Then we compute the transitive closure. If we infer $x \succ_v x$ for some variable x then there is a strict cycle in C. Because $\succ_v$ is a partial ordering and not a quasi-ordering there is no equivalence relation $\approx_v$ to take into account.

Hence, the test for strict cycles is considerably easier than for standard cycles. This alone justifies the addition of supplementary inequality constraints. As they are of the form $x \,\varrho\, t$ they are rather cheap. They do not lead to additional branching because of decomposition. The generation of supplementary equality constraints makes more knowledge explicit that is only implicit. This is not only useful for the search (the satisfiability test becomes a bit faster), but also a requirement for the use in confluence trees (see Section 6.4), an important application.

Finally, the influence of heuristics h_{Sel} and h_{Ord} must not be underestimated. For h_{Ord} the experiences of Section 4.2 suggest to consider the disjunctions introduced by the new decomposition rules from left to right. For h_{Sel} we found the following settings quite useful. First select atoms that have a variable as one of their sides. Then select an atom with the smallest branching factor. In our experiments, this simple scheme performs better than more elaborate ones that we have tested as well. Overall, the use of the improved h_{Sel} speeds up the test by more than one order of magnitude.

A rather unexpected result of the combination of improvements is that the cycle tests only rarely can prune the search tree. As is evident from the proof of Lemma 5.10, adding supplementary consequences by transitivity helps to shorten possible cycles. Strict cycles of length 2 can be detected by using the LPO extended by S: Consider the strict cycle established by atoms $A_1 \equiv x_1\, \varrho_1\, t_1[x_2]_{p_1}$ and $A_2 \equiv x_2\, \varrho_2\, t_2[x_1]_{p_2}$. Because the cycle is strict, A_1 is not of the form $x_1 \geq x_2$, hence $x_1 \succ^{A_1}_{\text{lpo}} x_2$ holds. By Corollary 5.2, we know that $A_1 \wedge A_2$ is unsatisfiable. It is therefore sufficient to test only solved forms for cycles. For the correctness of the test this is still necessary as there may be strict cycles of length 3 or more. The ommission of a part of the cycle tests leads not only to a simplification of the algorithm, but it speeds it up even further. Without the improvements of this section, the costs for cycle tests are dominant, with the improvements they are nearly negligible.

5.3 Sufficient tests for the unsatisfiability of KBO constraints

Compared to LPO, it took 10 years longer to show that the satisfiability of KBO constraints is decidable. In [KV00] Korovin and Voronkov developed a decision procedure by an elaborate reduction to linear arithmetic, the quantifier-free fragment of Presburger arithmetic. Shortly thereafter they refined the method resulting in an NP-algorithm [KV01]. However, this result is of more theoretical interest. Even the improved method is too involved to be implemented with reasonable effort. It is highly unlikely that it would run in reasonable time. To our knowledge up-to-now there exists no implementation of a decision procedure for KBO constraints[1]. We restrict our discussion to the common case that constraints are only conjunctions of atomic constraints. The extension to the general case is straightforward.

The KBO is parameterized by a weight function φ and a precedence $>_{\mathcal{F}}$. By its definition (see p. 54), it is a lexicographic combination of the weight-based ordering induced by φ, the precedence $>_{\mathcal{F}}$ applied to the top-symbols, and the lexicographic extension of itself applied to the arguments. The variable condition is only necessary to establish stability against substitutions. Hence, for checking satisfiability of KBO constraints, we have to focus on the weight-based ordering.

Function φ establishes a homomorphism from terms to the naturals. We can analyze the effect of applying a substitution to the weights of the resulting terms with the help of the following function ϕ:

$$\begin{aligned} \phi(x) &= x \\ \phi(f(t_1, \ldots, t_n)) &= \varphi(f) + \phi(t_1) + \cdots + \phi(t_n) \end{aligned}$$

For example, if $\phi(t) = 4+2\cdot x+y$, then we know that $\varphi(\sigma(t)) = 4+2\cdot\varphi(\sigma(x))+\varphi(\sigma(y))$. The following lemma can easily be shown by induction on the structure of terms.

[1] The work in [KV01] was not implemented, which is confirmed by Konstantin Korovin (personal e-mail, December 2003); and we have not heard of any implementation effort since then.

LEMMA 5.11 *Let* $\mathrm{Var}(t) = \{x_1, \dots, x_k\}$ *and* $w_i = \varphi(\sigma(x_i))$ *for* $i = 1, \dots, k$. *Then* $\varphi(\sigma(t)) = \phi(t)(w_1, \dots, w_k)$. □

With the help of ϕ we can define the following mapping Φ from constraints to conjunctions of linear Diophantine equations and inequations:

$$\begin{aligned}\Phi(s = t) &= \phi(s) - \phi(t) = 0\\ \Phi(s > t) &= \phi(s) - \phi(t) \geq 0\\ \Phi(s \geq t) &= \phi(s) - \phi(t) \geq 0\\ \Phi(C_1 \wedge \dots \wedge C_n) &= \Phi(C_1) \wedge \dots \wedge \Phi(C_n)\end{aligned}$$

The idea of this definition is simple: Let θ be a solution to the constraint of interest. Then for an equality constraint $s = t$ the weights of $\theta(s)$ and $\theta(t)$ have to be identical. For inequality constraints $s > t$ or $s \geq t$ the weight of $\theta(s)$ has to be greater than or equal to the weight of $\theta(t)$. Furthermore, we want to describe that the weight of $\theta(x)$ is strictly positive for each variable x contained in the constraint:

$$\mathrm{Pos}(C) = x_1 > 0 \wedge \dots \wedge x_k > 0 \qquad \text{if } \mathrm{Var}(C) = \{x_1, \dots, x_k\}$$

THEOREM 5.2 *Let* $\succ$ *be the KBO for* φ *and* $>_{\mathcal{F}}$. *If constraint* C *is satisfiable with respect to* $\succ$ *then the system of linear Diophantine equations and inequations* $\Phi(C) \wedge \mathrm{Pos}(C)$ *is satisfiable.*

PROOF Let $\theta \in \mathrm{Sol}(C)$, $\mathrm{Var}(C) = \{x_1, \dots, x_k\}$, and $w_i = \varphi(\theta(x_i))$ for $i = 1, \dots, k$. Then the assignment $x_i \mapsto w_i$, $i = 1, \dots, k$, satisfies any equation or inequation in $\Phi(C) \wedge \mathrm{Pos}(C)$: Consider equation $\phi(s) - \phi(t) = 0$ originating from constraint $s = t$. Substitution θ is a solution of the constraint, therefore $\theta(s) \equiv \theta(t)$, which implies $\varphi(\theta(s)) = \varphi(\theta(t))$. By Lemma 5.11 it follows that $\phi(s)(w_1, \dots, w_k) = \phi(t)(w_1, \dots, w_k)$, the assignment satisfies the equation. An inequation $\phi(s) - \phi(t) \geq 0$ originates from an inequality constraint $s \varrho t$, $\varrho \in \{>, \geq\}$. As θ is a solution, $\theta(s) \succeq \theta(t)$, hence $\varphi(\theta(s)) \geq \varphi(\theta(t))$. By Lemma 5.11, $\phi(s)(w_1, \dots, w_k) \geq \phi(t)(w_1, \dots, w_k)$, which means that the assignment satisfies the inequation. Finally, a property of φ is that $\varphi(t) > 0$ for any term t, which means that $w_i > 0$ for all i. Hence, the assignment satisfies the inequations of $\mathrm{Pos}(C)$. □

In contraposition, this gives a sufficient test for the unsatisfiability of KBO constraints: If $\Phi(C) \wedge \mathrm{Pos}(C)$ is unsatisfiable, then constraint C is unsatisfiable as well. The satisfiability of systems of linear Diophantine equations and inequations can be checked for example by the Omega-library [Pug92]. The result is an NP-algorithm. By giving up the requirement of integer solutions a polynomial time implementation becomes possible. However, this weakens the test and the Omega-library usually performs well in practice. The connections of KBO constraints and linear Diophantine equations are very close. In [KV00] a reverse translation from linear Diophantine equations to KBO constraints is given.

EXAMPLE 5.7 *Let* φ *return 1 for any symbol and let* $>_{\mathcal{F}}$ *be some arbitrary total precedence. Consider* $C_1 \equiv a \geq x \wedge x > b \wedge x \geq g(y) \wedge y > g(z)$. *Then* $\Phi(C_1)$ *returns*

$1 - x \geq 0 \wedge -1 + x \geq 0 \wedge -1 + x - y \geq 0 \wedge -1 + y - z \geq 0$. *This system of Diophantine inequations is satisfiable in the integers (e. g. with* $x \mapsto 1$*,* $y \mapsto -1$*,* $z \mapsto -2$*), but not in the naturals. However, there are no terms with negative weight! Such solutions are excluded by the inequations of* $\mathrm{Pos}(C_1)$*, the conjunction* $\Phi(C_1) \wedge \mathrm{Pos}(C_1)$ *is unsatisfiable.*

Consider $C_2 \equiv g(f(x,x)) > f(y,y) \wedge f(y,y) \geq h(f(x,x))$*. Then* $\Phi(C_2) \wedge \mathrm{Pos}(C_2)$ *is* $1 + 2x - 2y \geq 0 \wedge -1 - 2x + 2y \geq 0 \wedge x > 0 \wedge y > 0$*. Any solution of this system has to satisfy* $2y = 2x + 1$*. Therefore, there are solutions in the rationals (e. g.* $x \mapsto 1$*,* $y \mapsto 3/2$*), but not in the integers. The weight function* φ *maps into the naturals. Hence, there are no term bindings for the variables that correspond to the rational solutions, the ordering constraint is unsatisfiable.*

In case of $\phi(s) = \phi(t)$, the inequation $\phi(s) - \phi(t) \geq 0$ does not contribute any useful information. For any solution θ terms $\theta(s)$ and $\theta(t)$ have the same weight. If s and t have identical top-symbols, the ordering relation between $\theta(s)$ and $\theta(t)$ is decided by the lexicographic comparison of the arguments. For this case, we can therefore use the following variation of Φ which is inspired by the work of Hurd [Hur03]. Let $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv f(t_1, \ldots, t_n)$ with $\phi(s) = \phi(t)$. If $s \equiv t$ then $\Phi_{\mathrm{H}}(s > t) = \bot$ and $\Phi_{\mathrm{H}}(s \geq t) = \top$. If $s \not\equiv t$ then $\Phi_{\mathrm{H}}(s \,\varrho\, t) = \Phi_{\mathrm{H}}(s_i \geq t_i)$ if $s_j \equiv t_j$ for all $j < i$, $s_i \not\equiv t_i$ and $\varrho \in \{>, \geq\}$. In all other cases Φ_{H} is identical to Φ.

Theorem 5.3 *Let* $\succ$ *be the KBO for* φ *and* $>_{\mathcal{F}}$*. If constraint* C *is satisfiable with respect to* $\succ$ *then the system of linear Diophantine equations and inequations* $\Phi_{\mathrm{H}}(C) \wedge \mathrm{Pos}(C)$ *is satisfiable.*

Proof Let θ be a solution of C, $\mathrm{Var}(C) = \{x_1, \ldots, x_k\}$, and $w_i = \varphi(\theta(x_i))$ for $i = 1, \ldots, k$. Consider the induced assignment $x_i \mapsto w_i$, $i = 1, \ldots, k$. We concentrate on the difference between Φ and Φ_{H}. We show by induction on $|s| + |t|$ that $\theta(s) \succcurlyeq \theta(t)$ implies that the inequation delivered by $\Phi_{\mathrm{H}}(s \,\varrho\, t)$, $\varrho \in \{>, \geq\}$, is satisfied by the assignment. If $\phi(s) = \phi(t)$, $s \equiv f(s_1, \ldots, s_n)$, and $t \equiv f(t_1, \ldots, t_n)$ then either $s \equiv t$ or $s \not\equiv t$. In the first case, $\Phi_{\mathrm{H}}(s \geq t)$ returns the trivially satisfiable $\top$. As C is satisfiable it does not contain an atom $s > t$. If $s \not\equiv t$ then $\Phi_{\mathrm{H}}(s \,\varrho\, t) = \Phi_{\mathrm{H}}(s_i \geq t_i)$ where $s_j \equiv t_j$ for $j < i$. We know by Lemma 5.11 that $\varphi(\theta(s)) = \varphi(\theta(t))$. The top-symbols of s and t are equal. Therefore, the value of $\theta(s) \succcurlyeq \theta(t)$ is determined by the lexicographic comparison of the arguments. As $s_j \equiv t_j$ for $j < i$ it follows that $\theta(s_i) \succcurlyeq \theta(t_i)$ and induction hypothesis applies. If $\phi(s) \neq \phi(t)$ or s and t have different top-symbols then $\Phi_{\mathrm{H}}(s \,\varrho\, t)$ is $\phi(s) - \phi(t) \geq 0$. By Lemma 5.11, $\varphi(\theta(s)) \geq \varphi(\theta(t))$ implies $\phi(s)(w_1, \ldots, w_k) \geq \phi(t)(w_1, \ldots, w_k)$, which means that the assignment satisfies the inequation. □

With this modification the test for unsatisfiability becomes stronger.

Example 5.8 *A typical example where* Φ_{H} *is stronger than* Φ *is related to the commutativity axiom C. Let* φ *return 1 for any symbol and* $C \equiv x + y > y + x \wedge y \geq g(x)$*. Then* $\Phi(C)$ *is* $0 \geq 0 \wedge -1 - x + y \geq 0$ *which is clearly satisfiable. On the other hand,* $\Phi_{\mathrm{H}}(C)$ *is* $x - y \geq 0 \wedge -1 - x + y \geq 0$ *which is unsatisfiable.*

As determining an mgu of two terms is rather cheap, it is reasonable to preprocess equality constraints using unification. Furthermore, we can use ordering comparisons as a filter for the translation of inequality constraints. If we do not restrict the outcome to conjunctions of linear Diophantine inequations, it is possible to capture more information in the translation process. Disjunctions allow us to distinguish for constraint $s \geq t$ the two cases $s > t$ and $s = t$. If neither side is a variable this usually leads to a better translation. If at least one side is a variable, this is not advisable as in this case the translation is not improved. However, it introduces then unnecessary disjunctions which seem difficult to handle for the Omega-library.[2] For constraint $s > t$ even more cases are possible if s and t have the same top-symbol.

$$\begin{aligned}
\Phi'(s = t) &= \Phi_{\text{uni}}(s,t) \\
\Phi'(s > t) &= \begin{cases} \bot & \text{if } t \succcurlyeq s \\ \Phi_{\text{gt}}(s,t) & \text{otherwise} \end{cases} \\
\Phi'(s \geq t) &= \begin{cases} \phi(s) - \phi(t) \geq 0 & \text{if } s \in \mathcal{V} \text{ or } t \in \mathcal{V} \\ \Phi'(s > t) \vee \Phi'(s = t) & \text{otherwise} \end{cases} \\
\Phi'(C_1 \wedge \ldots \wedge C_n) &= \Phi'(C_1) \wedge \ldots \wedge \Phi'(C_n)
\end{aligned}$$

Function Φ_{uni} tests whether the arguments are unifiable and in the positive case relates the weights of bound variables to the weights of their bindings.

$$\Phi_{\text{uni}}(s,t) = \begin{cases} \bigwedge_{x \in \text{dom}(\sigma)} \phi(x) - \phi(\sigma(x)) = 0 & \text{if } \sigma = \text{mgu}(s,t) \\ \bot & \text{if } s \text{ and } t \text{ are not unifiable} \end{cases}$$

Depending on the top-symbols, function Φ_{gt} can generate stronger inequations. If they are identical, either the weight of s is greater than the weight of t or the weights are equal. Then there are n sub-cases reflecting the lexicographic comparison of the arguments.

$$\Phi_{\text{gt}}(s,t) = \begin{cases} \phi(s) - \phi(t) > 0 & \text{if } \text{top}(t) >_{\mathcal{F}} \text{top}(s) \\ \phi(s) - \phi(t) > 0 \ \vee & \text{if } \text{top}(s) = \text{top}(t) \\ \quad \phi(s) - \phi(t) = 0 \wedge \Phi_{\text{lex}}(\text{arg}(s), \text{arg}(t)) & \text{and } s \notin \mathcal{V} \\ \phi(s) - \phi(t) \geq 0 & \text{otherwise} \end{cases}$$

Function Φ_{lex} describes the lexicographic comparison of the argument lists. If there are no arguments left the result is $\bot$. Otherwise, either the first two arguments determine

[2] Consider the following example which we extracted during analyzing some irritating behavior of the Omega-library. Let $C_1 \equiv x_1 \geq x_2$ and $C_{i+1} \equiv C_i \wedge C_1$, i. e., C_i contains i copies of $x_1 \geq x_2$. Without the special treatment of variables in $\Phi'(s \geq t)$, the Omega-library needs about 370 MByte to represent the translation of C_{15} and about 75 seconds to determine satisfiability. Going from C_i to C_{i+1}, the memory requirement roughly doubles and the running time develops even worse. Apparently, the internal simplification mechanisms of the Omega-library do not handle such cases well.

the outcome or they have to be equal. The latter case is handled by function Φ_{eq}. If the first two arguments are unifiable, we encode that their instances have to be equal and call Φ_{lex} with the remaining arguments instantiated by the mgu of the first arguments. Otherwise, the result is $\bot$.

$$\Phi_{\mathrm{lex}}(\overline{s_n}, \overline{t_n}) = \begin{cases} \Phi'(s_1 > t_1) \vee \Phi_{\mathrm{eq}}(\overline{s_n}, \overline{t_n}) & \text{if } n > 0 \\ \bot & \text{otherwise} \end{cases}$$

$$\Phi_{\mathrm{eq}}(\overline{s_n}, \overline{t_n}) = \begin{cases} \Phi'(s_1 = t_1) \wedge \\ \quad \Phi_{\mathrm{lex}}(\sigma(s_2 \dots s_n), \sigma(t_2 \dots t_n)) & \text{if } \sigma = \mathrm{mgu}(s_1, t_1) \\ \bot & \text{if } s_1 \text{ and } t_1 \text{ are not unifiable} \end{cases}$$

THEOREM 5.4 *Let $\succ$ be the KBO for φ and $>_{\mathcal{F}}$. If constraint C is satisfiable with respect to $\succ$ then $\Phi'(C) \wedge \mathrm{Pos}(C)$ is satisfiable.*

PROOF By induction on $|C|$. Let $\theta \in \mathrm{Sol}(C)$, $\mathrm{Var}(C) = \{x_1, \dots, x_k\}$, and $w_i = \varphi(\theta(x_i))$ for $i = 1, \dots, k$. Then the assignment $x_i \mapsto w_i$, $i = 1, \dots, k$, satisfies $\Phi'(C) \wedge \mathrm{Pos}(C)$. Inequations in $\mathrm{Pos}(C)$ are trivially satisfied as $w_i > 0$ for all $i = 1, \dots, k$. The cases where C is a conjunction or $s \geq t$ is replaced by $s = t \vee s > t$ are trivial. The case where $\Phi'(s \geq t)$ delivers $\phi(s) - \phi(t) \geq 0$ is identical to the discussion in the proof of Theorem 5.2. Consider constraint $s = t$. As θ is a solution, we have $\theta(s) \equiv \theta(t)$, hence $\Phi_{\mathrm{uni}}(s, t) \neq \bot$ and $\theta \gtrsim \sigma$, $\sigma = \mathrm{mgu}(s, t)$. Therefore, $\theta(x) \equiv \theta(\sigma(x))$ and by Lemma 5.11 $\phi(x)(w_1, \dots, w_k) = \varphi(\theta(x)) = \varphi(\theta(\sigma(x))) = \phi(\sigma(x))(w_1, \dots, w_k)$ for all $x \in \mathrm{dom}(\sigma)$. The assignment satisfies the conjunction returned by $\Phi_{\mathrm{uni}}(s, t)$. Consider constraint $s > t$. As C is satisfiable, $t \not\succeq s$ and $\Phi'(s > t) = \Phi_{\mathrm{gt}}(s, t)$. We have $\theta(s) \succ \theta(t)$. If $\varphi(\theta(s)) > \varphi(\theta(t))$ then $\phi(s)(w_1, \dots, w_k) > \phi(t)(w_1, \dots, w_k)$ by Lemma 5.11, hence in all three cases the result of $\Phi_{\mathrm{gt}}(s, t)$ is satisfied by the assignment. If $\varphi(\theta(s)) = \varphi(\theta(t))$ we have to distinguish the three cases. The case $\mathrm{top}(t) >_{\mathcal{F}} \mathrm{top}(s)$ cannot happen, otherwise were $\theta(t) \succ \theta(s)$. In case of $\mathrm{top}(s) = \mathrm{top}(t)$ and $s \notin \mathcal{V}$, the result of $\theta(s) \succ \theta(t)$ is determined by the lexicographic comparison of the arguments. Let $s \equiv f(s_1, \dots, s_n)$ and $t \equiv f(t_1, \dots, t_n)$. Consider the constraint delivered by $\Phi_{\mathrm{gt}}(s, t)$ which is $\phi(s) - \phi(t) > 0 \vee \phi(s) - \phi(t) = 0 \wedge \Phi_{\mathrm{lex}}(s_1 \dots s_n, t_1 \dots t_n)$. Because of Lemma 5.11 we know that the assignment does not satisfy the first inequation, but satisfies the equation. Therefore, we have to show that the assignment satisfies the formula returned by $\Phi_{\mathrm{lex}}(s_1 \dots s_n, t_1 \dots t_n)$. We know that there is some i, $1 \leq i \leq n$, such that $\theta(s_j) \equiv \theta(t_j)$ for $j < i$ and $\theta(s_i) \succ \theta(t_i)$. If $i = 1$, then $\theta(s_1) \succ \theta(t_1)$ and by induction hypothesis $\Phi'(s_1 > t_1)$ is satisfied. Otherwise, $\theta(s_1) \equiv \theta(t_1)$, which implies that $\Phi'(s_1 = t_1)$ is satisfied. Furthermore, $\theta \gtrsim \sigma$, where $\sigma = \mathrm{mgu}(s_1, t_1)$. By iterating the argumentation it is easy to see that therefore the formula returned by $\Phi_{\mathrm{lex}}(\sigma(s_2 \dots s_n), \sigma(t_2 \dots t_n))$ is satisfied by the assignment induced by θ. For the third case, where $\Phi'(s > t)$ returns $\phi(s) - \phi(t) \geq 0$, it is clear that $\varphi(\theta(s)) = \varphi(\theta(t))$ implies that the assignment satisfies the inequation. $\square$

The more elaborate translation results in a better description of the weight relations induced by the ordering constraints and therefore in a stronger test.

EXAMPLE 5.9 *Let* φ *return 1 for any symbol and let the precedence be* $f >_{\mathcal{F}} g >_{\mathcal{F}} h >_{\mathcal{F}} a$. *Consider first constraint* $C_1 \equiv h(a) > g(x)$. *Functions* Φ *and* Φ_{H} *return both* $1 - x \geq 0$ *which has the satisfying assignment* $x \mapsto 1$. *Function* Φ' *delivers* $1 - x > 0$ *which is unsatisfiable together with* $\mathrm{Pos}(C_1) = x > 0$.

Let C_2 *be the constraint* $f(x,x,y) = f(g(a), y', y')$. *The result of function* Φ *or* Φ_{H} *is* $-2+2x+y-2y'=0$. *Function* Φ' *computes the* mgu *which is* $\{x \mapsto g(a), y \mapsto g(a), y' \mapsto g(a)\}$ *and returns the more precise conjunction* $-2+x=0 \wedge -2+y=0 \wedge -2+y'=0$.

More complicated is the translation of $C_3 \equiv f(z, h(z), h(g(y))) > f(g(a), g(g(x)), z)$ *by* Φ'. *We get as intermediate result* $-1 - x + y + z > 0 \vee (-1 - x + y + z = 0 \wedge \Phi'(z > g(a)) \vee (\Phi'(z = g(a)) \wedge (\Phi'(h(g(a)) > g(g(x))) \vee \bot)))$. *Note, that the binding of* z *is propagated and that* Φ_{eq} *returns* $\bot$ *instead of considering the third arguments as* $h(g(a))$ *and* $g(g(x))$ *are not unifiable. The final result is* $-1 - x + y + z > 0 \vee (-1-x+y+z=0 \wedge -2+z \geq 0 \vee (-2+z=0 \wedge (1-x>0 \vee \bot)))$. *As we already know from the analysis of* C_1, *inequation* $1-x>0$ *is unsatisfiable together with* $\mathrm{Pos}(C_3)$. *We can therefore simplify the result to* $-1-x+y+z>0 \vee -1-x+y+z=0 \wedge -2+z \geq 0$ *Functions* Φ *and* Φ_{H} *return for* C_3 *the inequation* $-1 - x + y + z \geq 0$.

Finally, we want to consider $C_4 \equiv a \geq z \wedge C_2 \wedge C_3$. *This requires that any solution of* C_4 *binds* z *to a term of weight 1. Therefore,* $\Phi'(C_4) \wedge \mathrm{Pos}(C_4)$ *is unsatisfiable, which can be checked easily by substituting the known values for* x, y, *and* z *into the simplified result of* $\Phi'(C_3)$. *On the other side, by using* Φ *or* Φ_{H}, *the unsatisfiability of* C_4 *remains undetected. Function* Φ' *achieves the necessary precision by distinguishing whether for* C_3 *the comparison of the weights suffices or not. In the latter case it then differentiates between the sub-cases of the lexicographic comparison.*

Nevertheless, all the translations presented so far lead only to sufficient tests. A simple example for which all tests fail to detect unsatisfiability is the following.

EXAMPLE 5.10 *Consider* $C \equiv x > y \wedge h(y) \geq h(x)$. *This ordering constraint is clearly unsatisfiable. However,* $\Phi(C) \wedge \mathrm{Pos}(C)$ *is* $x - y \geq 0 \wedge -x + y \geq 0 \wedge x > 0 \wedge y > 0$. *Functions* Φ_{H} *and* Φ' *give for* C *essentially the same result as function* Φ. *This system of Diophantine inequations has an infinite number of solutions mapping* x *and* y *to the same positive natural number.*

This shows the main weakness of the simple translation approaches, they capture mainly the weight-based part of KBO. As soon as one side of an inequality constraint is a variable and the other side is not, a decision procedure has to consider the top-symbols of possible bindings of the variable. Some other properties have to be encoded, too. For example, that there exists at least a certain number of different terms of the same weight. The method of [KV01] uses therefore a far more elaborate translation that produces significantly larger and more complicated arithmetic constraints. It remains to be seen if the more general work recently presented in [ZSM05] allows the derivation of a simpler approach.

5.4 A generic test for ground reduction orderings

The ordering specific tests that we have presented in the previous sections have two disadvantages: First, they are all NP-methods, which means that we have to admit exponential runtime behavior, at least occasionally. Second, they are tailored very closely to the ordering. So it is impossible to adapt them with small modifications to other orderings. Of course, if we want to have decision procedures for the constraint satisfiability problem we cannot avoid exponential behavior by NP-completeness of the problem – unless P=NP. However, it is often enough to have a sufficient test for the unsatisfiability of ordering constraints. Our aim is a test that is reasonably precise and efficient, easy to implement, and covers ground reduction orderings in a generic way. Especially, we want to have a polynomial time algorithm.

The input to our test is a conjunction C of atomic constraints. The main idea is to saturate C by using properties of the ordering to derive new atomic constraints from existing ones. If a constraint is derived that is obviously unsatisfiable, such as $s > s$, we know that C is unsatisfiable as well. The properties of the ordering are reflected by corresponding saturation rules. This allows us to use a modular design. Depending on the application and the information available we can modify the set of saturation rules. If we know the type of the ordering we can extend this set by rules using properties of the specific ordering scheme, such as LPO or KBO (see Section 5.4.1). This includes the use of precedence $>_{\mathcal{F}}$ or weight function φ. On the other side, if in some circumstances the ordering at hand is not total on ground terms we can omit the corresponding rule.

For our standard application we require that $\succ$ is a ground reduction ordering. Hence, it is e. g. irreflexive, transitive, monotonic, and total on ground terms. These properties are sufficient in the following example.

Example 5.11 *Let the input constraint C be $g(x) > g(y) \wedge h(y) \geq h(x)$. By ground totality of $\succ$ we can derive that $x{>}y$ (ground instances of x and y have to be comparable and $y{\geq}x$ would imply $g(y){\geq}g(x)$ in violation with $g(x){>}g(y)$, cf. Lemma 5.13). By the monotonicity rule we then derive the new constraint $h(x){>}h(y)$, then by the transitivity rule $h(x) > h(x)$. The irreflexivity rule detects this atomic constraint unsatisfiable, so we know that the original constraint is unsatisfiable as well.*

Except the first step, this example may look too simplistic. However, for this example, all methods of Section 5.3 fail to detect unsatisfiability – even if we start with the easier constraint $x > y \wedge h(y) \geq h(x)$ (see Example 5.10).

Related to our method is the work of Plaisted [Pla93] and of Johann and Socher-Ambrosius [JSA94]. They are essentially interested in the problem whether or not for a given constraint C there exists a simplification ordering $\succ$ and a ground substitution σ such that σ satisfies C with respect to $\succ$. Hence, in contrast to other approaches, the ordering $\succ$ is not part of the input, its existence is part of the output. Our problem is different, we have a fixed ordering which is known. We are interested in the negative outcomes of the test. Hence, we can extend the approach and adapt it to our needs. For example, if $s \succ t$ we can add the constraint $s > t$ without changing satisfiability. This may enable the use of other rules and thus may contribute to an unsatisfiability

proof. Furthermore, our ordering is total on ground terms, which is not handled by the tests in [Pla93] and [JSA94].

Preprocessing

The input constraints usually have many subterms in common. So it is rewarding to identify them in a preprocessing phase, which furthermore simplifies the main saturation process. We describe the preprocessing in an abstract way by introducing a set $K = \{c_1, c_2, \ldots\}$ of new constants to represent the different subterms in the original problem. This formulation is inspired by modern treatments of congruence closure algorithms [Kap97, BTV03].

We call a term *flat*, if it is a variable x or of the form $f(c_1, \ldots, c_n)$, that is, an operator applied to new constants only. Let $\mathfrak{C}$ be a conjunction of atomic constraints and $\mathfrak{D}$ a rewrite system, which consists of rules $t \to c$ where t is a flat term and $c \in K$. Initially, $\mathfrak{D}$ is empty and $\mathfrak{C}$ contains the original constraint C which we want to test for unsatisfiability. The following two transformation rules assign new constants as names to subterms and propagate them through the problem.

1. Naming:
$$\langle \mathfrak{C}[t], \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C}[c], \mathfrak{D} \cup \{t \to c\} \rangle$$
if t is a flat term in $\mathfrak{D}$-normal form and c a new constant that is not present in $\mathfrak{C}$ or $\mathfrak{D}$.

2. Propagation:
$$\langle \mathfrak{C}[t], \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C}[c], \mathfrak{D} \rangle$$
if $t \to c$ is in $\mathfrak{D}$.

To apply the transformation rules, we perform a leftmost-innermost traversal of the terms. For each subterm we first try to apply Propagation and replace the subterm by its corresponding constant. If that is not possible, we apply Naming and introduce a new constant as name for the subterm. After the transformation, $\mathfrak{C}$ contains only atomic constraints of the form $c \,\varrho\, c'$ where $c, c' \in K$. Each subterm t of C is then uniquely represented by a new constant c. Hence $\mathfrak{D}$ defines a function $T_{\mathfrak{D}} : K \to \mathrm{Term}(\mathcal{F}, \mathcal{V})$ such that $T_{\mathfrak{D}}(c)$ is the subterm that c represents. For the new representation $\langle \mathfrak{C}, \mathfrak{D} \rangle$ we have to adapt the definition that a ground substitution σ satisfies a constraint C with respect to an ordering $\succ$.

Definition 5.9 *Let $\langle \mathfrak{C}, \mathfrak{D} \rangle$ be the result of the transformation rules (i. e., neither* Naming *nor* Propagation *are applicable). Substitution σ* satisfies *the constraints $c{>}c'$, $c{\geq}c'$, or $c{=}c'$ with respect to $\succ$ iff $\sigma(T_{\mathfrak{D}}(c)) \succ \sigma(T_{\mathfrak{D}}(c'))$, $\sigma(T_{\mathfrak{D}}(c)) \succeq \sigma(T_{\mathfrak{D}}(c'))$, or $\sigma(T_{\mathfrak{D}}(c)) \equiv \sigma(T_{\mathfrak{D}}(c'))$. Substitution σ* satisfies *$\langle \mathfrak{C}, \mathfrak{D} \rangle$ with respect to $\succ$ iff σ satisfies any constraint $c \,\varrho\, c'$ contained in $\mathfrak{C}$ with respect to $\succ$.*

Lemma 5.12 *Let $\langle \mathfrak{C}, \mathfrak{D} \rangle$ be the result of preprocessing C. Then* $\mathrm{GSub}(C) = \mathrm{GSub}(\mathfrak{D})$ *and* $\mathrm{Var}(\mathfrak{C}) = \varnothing$. *Furthermore, σ satisfies C with respect to $\succ$ if, and only if, σ satisfies*

$\langle \mathfrak{C}, \mathfrak{D} \rangle$ *with respect to* $\succ$*. The transformation needs a number of steps that is linear in the size of* C*.*

PROOF During the transformation no new variables are introduced and all variables occurring in C are transferred to $\mathfrak{D}$. Hence, $\mathrm{Var}(\mathfrak{C}) = \varnothing$, $\mathrm{Var}(\mathfrak{D}) = \mathrm{Var}(C)$ and $\mathrm{GSub}(C) = \mathrm{GSub}(\mathfrak{D})$. The transformation preserves the structures of the constraints. Only the representation of terms is modified. Therefore, $s \varrho t$ is in C iff $c \varrho c'$ is in $\mathfrak{C}$ with $T_{\mathfrak{D}}(c) \equiv s$ and $T_{\mathfrak{D}}(c') \equiv t$. This implies that satisfiability is preserved. For each subterm in C either rule NAMING or rule PROPAGATION is applied once. Hence, a linear number of steps is performed. □

EXAMPLE 5.12 *Consider constraint* $f(f(x)) > g(f(x))$*. The first step is to give* x *the name* c_1 *and to record this by adding the rule* $x \to c_1$ *to* $\mathfrak{D}$*. Similarly, we add the rules* $f(c_1) \to c_2$ *and* $f(c_2) \to c_3$*. Next, we traverse the second term. We first propagate already introduced names and replace* x *by* c_1 *and* $f(c_1)$ *by* c_2*. Then for the remaining term we add the rule* $g(c_2) \to c_4$*. The final form of the constraint is* $c_3 > c_4$*. It is equivalent to* $f(f(x)) > g(f(x))$ *as* $T_{\mathfrak{D}}(c_3) \equiv f(f(x))$ *and* $T_{\mathfrak{D}}(c_4) \equiv g(f(x))$*.*

The example is atypical in that it reduces the problem size from 6 to 4. Typically, the input to the test is much larger and the transformation reduces the problem size by a factor of at least 2 to about 4.

The saturation process

After the preprocessing we start the saturation process. It is based on the following saturation rules, which are suitable for all ground reduction orderings. We assume that a rule is only performed if the derived constraint is new to $\mathfrak{C}$. Recall that we regard equality constraints as symmetric and do not distinguish $c = c'$ and $c' = c$. Hence, $c = c'$ is in $\mathfrak{C}$ iff $c' = c$ is in $\mathfrak{C}$. Furthermore, we assume that $\mathfrak{C}$ contains for each constant c the constraint $c = c$. This does not affect the set of solutions and allows us to simplify conditions such as "either $\mathfrak{C}$ contains $c_1 = c_2$, or $c_1 \equiv c_2$". Some rules use ordering $\sqsupset$ on constraint symbols. Recall that $> \sqsupset \geq \sqsupset =$.

1. ORDERING:
$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c > c'\}, \mathfrak{D} \rangle$$
if $T_{\mathfrak{D}}(c) \succ T_{\mathfrak{D}}(c')$.

2. IRREFLEXIVITY:
$$\langle \mathfrak{C} \cup \{c > c\}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \bot, \mathfrak{D} \rangle$$

3. TRANSITIVITY:
$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c_1 \varrho c_3\}, \mathfrak{D} \rangle$$
if $\mathfrak{C}$ contains $c_1 \varrho_1 c_2$ and $c_2 \varrho_2 c_3$, and $\varrho = \max_{\sqsupset}\{\varrho_1, \varrho_2\}$.

4. Monotonicity:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c\,\varrho\,c'\}, \mathfrak{D} \rangle$$

if $\mathfrak{D}$ contains the rules $f(c_1,\ldots,c_n) \rightarrow c$ and $f(c'_1,\ldots,c'_n) \rightarrow c'$, $\mathfrak{C}$ contains $c_i\,\varrho_i\,c'_i$ for all $i = 1,\ldots,n$, and $\varrho = \max_{\sqsupset}\{\varrho_i \mid i = 1,\ldots,n\}$.

5. Context:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c_i\,\varrho\,c'_i\}, \mathfrak{D} \rangle$$

if $\mathfrak{D}$ contains the rules $f(c_1,\ldots,c_n) \rightarrow c$ and $f(c'_1,\ldots,c'_n) \rightarrow c'$, $\mathfrak{C}$ contains $c\,\varrho\,c'$ with $\varrho \in \{>,\geq\}$, and $\mathfrak{C}$ contains $c_j = c'_j$ for $j = 1,\ldots,n$ with $j \neq i$.

6. Decomposition:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c_i = c'_i\}, \mathfrak{D} \rangle$$

if $\mathfrak{D}$ contains the rules $f(c_1,\ldots,c_n) \rightarrow c$ and $f(c'_1,\ldots,c'_n) \rightarrow c'$, and $\mathfrak{C}$ contains $c = c'$.

7. Clash:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \bot, \mathfrak{D} \rangle$$

if $\mathfrak{C}$ contains $c = c'$ and $T_{\mathfrak{D}}(c)$ is not unifiable with $T_{\mathfrak{D}}(c')$.

8. Strengthening:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c > c'\}, \mathfrak{D} \rangle$$

if $\mathfrak{C}$ contains $c \geq c'$ and $T_{\mathfrak{D}}(c)$ is not unifiable with $T_{\mathfrak{D}}(c')$.

9. Antisymmetry:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c = c'\}, \mathfrak{D} \rangle$$

if $\mathfrak{C}$ contains $c \geq c'$ and $c' \geq c$.

10. Absorption:

$$\langle \mathfrak{C} \cup \{c \geq c', c\,\varrho\,c'\}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c\,\varrho\,c'\}, \mathfrak{D} \rangle$$

if $\varrho \in \{>,=\}$.

Note that the saturation rules do not introduce new terms (i. e., constants), they merely add relations between existing ones. Applying substitutions as in [JSA94] can lead to an exponential growth of problem size. Only three rules take into account the term structure. Whereas Monotonicity works upwards and uses knowledge about the subterms, the rules Decomposition and Context work downwards and derive knowledge about the subterms. The transformation rules preserve satisfiability:

Lemma 5.13 *Let $\succ$ be a reduction ordering that is total on ground terms. Let $\langle \mathfrak{C}, \mathfrak{D} \rangle \Longrightarrow \langle \mathfrak{C}', \mathfrak{D} \rangle$ with one of the saturation rules and let $\sigma \in \mathrm{GSub}(\mathfrak{D})$. If σ satisfies $\langle \mathfrak{C}, \mathfrak{D} \rangle$ with respect to $\succ$, then σ satisfies $\langle \mathfrak{C}', \mathfrak{D} \rangle$ with respect to $\succ$.*

Proof This is a consequence of properties concerning the ground reduction ordering $\succ$ and the satisfiability of constraints. We consider the different rules in turn.

1. Ordering: For any substitution θ, relation $T_{\mathfrak{D}}(c) \succ T_{\mathfrak{D}}(c')$ implies $\theta(T_{\mathfrak{D}}(c)) \succ \theta(T_{\mathfrak{D}}(c'))$ by stability of $\succ$. Hence σ satisfies $c > c'$ with respect to $\succ$.
2. Irreflexivity: By assumption, $\succ$ is a reduction ordering, hence irreflexive. Therefore, $\mathrm{Sol}(T_{\mathfrak{D}}(c) > T_{\mathfrak{D}}(c)) = \varnothing = \mathrm{Sol}(\bot)$.
3. Transitivity: Simple consequence of the transitivity of $\succ$.
4. Monotonicity: Let $s \equiv T_{\mathfrak{D}}(c)$ and $t \equiv T_{\mathfrak{D}}(c')$, therefore $s \equiv f(s_1, \ldots, s_n)$, $t \equiv f(t_1, \ldots, t_n)$, and $s_i \equiv T_{\mathfrak{D}}(c_i)$ and $t_i \equiv T_{\mathfrak{D}}(c'_i)$ for $i = 1, \ldots, n$. Define $u_i \equiv \sigma(f(t_1, \ldots, t_i, s_{i+1}, \ldots, s_n))$ for $i = 0, \ldots, n$, hence $u_0 \equiv \sigma(s)$ and $u_n \equiv \sigma(t)$. If $\mathfrak{C}$ contains $c_i = c'_i$ then $\sigma(s_i) \equiv \sigma(t_i)$ and $u_{i-1} \equiv u_i$. If $\mathfrak{C}$ contains $c_i > c'_i$ then $\sigma(s_i) \succ \sigma(t_i)$ and $u_{i-1} \succ u_i$ by monotonicity of $\succ$. If $\mathfrak{C}$ contains $c_i \geq c'_i$ then either $\sigma(s_i) \equiv \sigma(t_i)$ and $u_{i-1} \equiv u_i$ or $\sigma(s_i) \succ \sigma(t_i)$ and $u_{i-1} \succ u_i$ by monotonicity of $\succ$. Transitivity of $\succ$ implies that either $u_0 \equiv u_n$ (if $\sigma(s_i) \equiv \sigma(t_i)$ for all $i = 1, \ldots, n$) or $u_0 \succ u_n$. Therefore, either $\sigma(s) \equiv \sigma(t)$ or $\sigma(s) \succ \sigma(t)$ holds; σ satisfies $s \varrho t$.
5. Context: Let $s \equiv T_{\mathfrak{D}}(c)$ and $t \equiv T_{\mathfrak{D}}(c')$, therefore $s \equiv f(s_1, \ldots, s_n)$, $t \equiv f(t_1, \ldots, t_n)$, and $s_i \equiv T_{\mathfrak{D}}(c_i)$ and $t_i \equiv T_{\mathfrak{D}}(c'_i)$ for $i = 1, \ldots, n$. Consider the case where ϱ is $>$. If σ satisfies $\langle \mathfrak{C}, \mathfrak{D} \rangle$ then $\sigma(s) \succ \sigma(t)$ and $\sigma(s_j) \equiv \sigma(t_j)$ for all $j \neq i$. Consider $\sigma(s_i)$ and $\sigma(t_i)$. As $\sigma \in \mathrm{GSub}(\mathfrak{D})$ both terms are ground terms and thus either identical or comparable by $\succ$. If $\sigma(t_i) \succcurlyeq \sigma(s_i)$ then $\sigma(t) \succcurlyeq \sigma(s)$ by monotonicity of $\succ$. This is a contradiction to $\sigma(s) \succ \sigma(t)$, hence $\sigma(s_i) \succ \sigma(t_i)$ must hold. The case where ϱ is $\geq$ is similar.
6. Decomposition: Let $s \equiv T_{\mathfrak{D}}(c)$ and $t \equiv T_{\mathfrak{D}}(c')$, therefore $s \equiv f(s_1, \ldots, s_n)$, $t \equiv f(t_1, \ldots, t_n)$, and $s_i \equiv T_{\mathfrak{D}}(c_i)$ and $t_i \equiv T_{\mathfrak{D}}(c'_i)$ for $i = 1, \ldots, n$. If σ satisfies $s = t$ then $\sigma(s) \equiv \sigma(t)$, hence $\sigma(s_i) \equiv \sigma(t_i)$ for all $i = 1, \ldots, n$.
7. Clash: If $T_{\mathfrak{D}}(c)$ is not unifiable with $T_{\mathfrak{D}}(c')$ then $\mathrm{Sol}(T_{\mathfrak{D}}(c) = T_{\mathfrak{D}}(c')) = \varnothing = \mathrm{Sol}(\bot)$.
8. Strengthening: If $T_{\mathfrak{D}}(c)$ is not unifiable with $T_{\mathfrak{D}}(c')$ then $\mathrm{Sol}(T_{\mathfrak{D}}(c) \geq T_{\mathfrak{D}}(c')) = \mathrm{Sol}(T_{\mathfrak{D}}(c) > T_{\mathfrak{D}}(c'))$.
9. Antisymmetry: $\sigma(T_{\mathfrak{D}}(c)) \succcurlyeq \sigma(T_{\mathfrak{D}}(c'))$ and $\sigma(T_{\mathfrak{D}}(c')) \succcurlyeq \sigma(T_{\mathfrak{D}}(c))$ imply that $\sigma(T_{\mathfrak{D}}(c)) \equiv \sigma(T_{\mathfrak{D}}(c'))$.
10. Absorption: All constraints in $\mathfrak{C}'$ are contained in $\mathfrak{C}$. So any solution of $\langle \mathfrak{C}, \mathfrak{D} \rangle$ is a solution of $\langle \mathfrak{C}', \mathfrak{D} \rangle$.

□

Let us consider Example 5.11 again and reformulate it in a more concrete way by using the introduced rules.

Example 5.13 *Let C be $g(x) > g(y) \wedge h(y) \geq h(x)$. Preprocessing results in $\mathfrak{C} \equiv c_2 > c_4 \wedge c_5 \geq c_6$ and $\mathfrak{D}$ contains the rules $x \to c_1$, $g(c_1) \to c_2$, $y \to c_3$, $g(c_3) \to c_4$, $h(c_3) \to c_5$, and $h(c_1) \to c_6$. Then the saturation derives $c_1 > c_3$ by rule* Context, *then $c_6 > c_5$ by rule* Monotonicity, *then $c_6 > c_6$ by rule* Transitivity, *and finally $\bot$ by rule* Irreflexivity. *Hence, C is unsatisfiable.*

To demonstrate the desired results for the asymptotic running time of our method in the following we give some implementation details. The input is a conjunction of atomic constraints C. The implementation of our method is based on three tables. The first one represents $\mathfrak{D}$. For each new constant c it offers the access to rule $f(c_1, \ldots, c_n) \rightarrow c$ and directly to $T_{\mathfrak{D}}(c)$. Furthermore, it provides a function *parent*(c, i), which for c gives all constants c' that have c as their i-th subterm in the left-hand side of their defining rule in $\mathfrak{D}$. This is important for the efficient implementation of some of the more complicated rules such as MONOTONICITY. After the first phase of the algorithm, in which we build up $\mathfrak{D}$ from C, the set of constants is fixed, say to $\{c_1, \ldots, c_N\}$. Then we can allocate for $\mathfrak{C}$ a quadratic table with $N \times N$ entries containing constraint symbols. The entries $\mathfrak{C}[c_i, c_i]$ are set to $=$, for the other entries we try to apply rule ORDERING. If this is done in a bottom-up way, the initialization of $\mathfrak{C}$ can be performed in $O(N^2)$ time, both for KBO and LPO. Then we insert the original constraints of C into $\mathfrak{C}$ and start the saturation with regard to the remaining rules.

In the third table, which we call $\mathfrak{L}$, for each modification of $\mathfrak{C}$ we add an entry with the old value, the new value, and the applied rule together with justifications. The size of $\mathfrak{L}$ is $2N^2$ in the worst case, as for each entry in $\mathfrak{C}$ at most two insertions can occur (first $\geq$, then either $>$ or $=$). Table $\mathfrak{L}$ provides several functionalities in a convenient way. First, with a simple index we can keep track for which constraints we already have applied all saturation rules, and for which this still has to be done. Second, if one of the rules IRREFLEXIVITY or CLASH applies, we can extract from $\mathfrak{L}$ a detailed justification why the original constraint is unsatisfiable. This is helpful not only for debugging purposes, but also for determining the subset of the atomic constraints that leads to unsatisfiability. We call this the *unsatisfiable kernel* of C. Determining the unsatisfiable kernel can be used to restrict some sets and therefore to accelerate some operations needed by the redundancy criteria of Section 6.5. Typically, the unsatisfiable kernel consists of about 30 % to 50 % of the atoms of C. Finally, $\mathfrak{L}$ facilitates an undo-mechanism. This is interesting, if we want to extend the constraint incrementally, and later retract such additions. With table $\mathfrak{L}$ we can do this in a stack-like manner, which is sufficient for example for the use in confluence trees (cf. Section 6.4). Furthermore, we could use the undo-mechanism of $\mathfrak{L}$ to handle disjunctive constraints by backtracking. However, we have not pursued this further as the resulting algorithm would need exponential time.

THEOREM 5.5 *Let C be a conjunction of atomic constraints. The algorithm based on the transformation and saturation rules is correct: If it derives $\langle \bot, \mathfrak{D} \rangle$ from $\langle C, \varnothing \rangle$, then C is unsatisfiable. It needs $O(N^2)$ space and runs in $O(N^3 + |C| + T_{\mathrm{Ord}}(N))$ time, where N is the number of distinct subterms in C and $T_{\mathrm{Ord}}(N)$ the time to determine the ordering relations between the N distinct subterms.*

PROOF By Lemma 5.12 and Lemma 5.13 and induction on the number of saturation steps the algorithm preserves satisfiability. As previously discussed, the space required for the three tables is in $O(N^2)$. To analyze the time requirements, we distinguish the preprocessing, the initialization of table $\mathfrak{C}$, the insertion of the constraints, and the actual saturation process. With appropriate indexing data structures, the construction

of $\mathfrak{D}$ can be performed in time $O(|C|)$. The initialization of $\mathfrak{C}$ with the ordering relations between the subterms is in $O(T_{\text{Ord}}(N))$. The time needed to insert the constraints of C into $\mathfrak{C}$ is in $O(|C|)$. For the saturation process, consider all pairs $\langle c, c' \rangle$ of new constants. For each pair at most two values are inserted into $\mathfrak{C}$ (first $\geq$, then either $>$ or $=$) which covers the work induced by rule Absorption. However, many more times values for $\langle c, c' \rangle$ may be computed by one of the other saturation rules. Rule Transitivity may compute $O(N)$ times a value for $\langle c, c' \rangle$, each time with constant costs. Rules Decomposition and Antisymmetry may compute once a value with constant costs each. Rules Monotonicity and Context may compute $O(n)$ times a value with costs in $O(n)$ each time, where n is bound by the maximal arity of a function symbol, hence constant. Rule Strengthening may be tested once with costs that are linear in N. Constraint $\perp$ can be derived by rules Irreflexivity and Clash which may be triggered by the insertion of a relation $c \varrho c'$ into $\mathfrak{C}$. They need either constant time or time that is linear in N. As there are $O(N^2)$ pairs and the costs per pair are in $O(N)$ for all saturation rules, the running time of the saturation is in $O(N^3)$. □

Note that for the runtime analysis it is important that table $\mathfrak{D}$ provides function $\mathit{parent}(c, i)$ in $O(1)$. Therefore, rules Monotonicity and Context can directly access the parents. Otherwise, they would have to search for them which would increase the computational costs to $O(N^2)$ for each pair $\langle c, c' \rangle$. This would imply that the test would need $O(N^4 + |C| + T_{\text{Ord}}(N))$ time. As already mentioned, for LPO and KBO, the orderings of interest, $O(T_{\text{Ord}}(N)) = O(N^2)$. Therefore, the test needs $O(N^3 + |C|)$ time in these cases. If the evaluation of the ordering is too expensive we can approximate it by $\succ_{\text{st}}$ thereby weakening the test. It is well known that the subterm ordering $\succ_{\text{st}}$ is contained in $\succ$ for any ground reduction ordering $\succ$.

5.4.1 Extensions for particular orderings

An important property of our unsatisfiability test is its generic nature, it is suitable for all ground reduction orderings. However, the modular design based on saturation rules permits us to extend the test for particular orderings. In the following, we give additional rules for the LPO and the KBO, the orderings that are most frequently used in practice.

Additional rules for LPO

For LPO we have the following additional rules, which closely reflect its definition:

1. Beta:
$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c > c'\}, \mathfrak{D} \rangle$$
if $\mathfrak{D}$ contains the rules $f(\dots) \to c$ and $g(c'_1, \dots, c'_n) \to c'$, $\mathfrak{C}$ contains $c > c'_i$ for all $i = 1, \dots, n$, and $f >_{\mathcal{F}} g$.

2. Gamma:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c > c'\}, \mathfrak{D} \rangle$$

if $\mathfrak{D}$ contains the rules $f(c_1, \dots, c_n) \to c$ and $f(c'_1, \dots, c'_n) \to c'$, there is some i with $1 \le i < n$ such that $\mathfrak{C}$ contains $c_i > c'_i$ and contains $c_j \, \varrho_j \, c'_j$ with $\varrho_j \in \{\ge, =\}$ for all $j = 1, \dots, i-1$ and contains $c > c'_k$ for all $k = i+1, \dots, n$.

Note that for rule Gamma case $i = n$ is excluded because it is already covered by rule Monotonicity.

Lemma 5.14 *Let $\succ$ be a ground-total LPO for precedence $>_{\mathcal{F}}$. Let $\langle \mathfrak{C}, \mathfrak{D} \rangle \Longrightarrow \langle \mathfrak{C}', \mathfrak{D} \rangle$ with* Beta *or* Gamma *and let $\sigma \in \mathrm{GSub}(\mathfrak{D})$. If σ satisfies $\langle \mathfrak{C}, \mathfrak{D} \rangle$ with respect to $\succ$, then σ satisfies $\langle \mathfrak{C}', \mathfrak{D} \rangle$ with respect to $\succ$. Extending the test by* Beta *and* Gamma *does not increase its asymptotic running time.*

Proof Both rules preserve satisfiability: Case Beta: Let $\sigma(T_{\mathfrak{D}}(c)) \equiv s \equiv f(s_1, \dots, s_n)$ and $\sigma(T_{\mathfrak{D}}(c')) \equiv t \equiv g(t_1, \dots, t_m)$. By assumption we have $f >_{\mathcal{F}} g$ and $s \succ t_i$ for all $i = 1, \dots, m$. Because $\succ$ is some LPO we have then $s \succ t$ by the case (β) in the definition of LPO. Case Gamma: Let $\sigma(T_{\mathfrak{D}}(c)) \equiv s \equiv f(s_1, \dots, s_n)$ and $\sigma(T_{\mathfrak{D}}(c')) \equiv t \equiv f(t_1, \dots, t_m)$. By assumption we have for all $j < i$ either $s_j \equiv t_j$ or $s_j \succ t_j$. Furthermore, $s_i \succ t_i$ and $s \succ t_k$ for all $k > i$. Let i_0 be the smallest index with $s_{i_0} \succ t_{i_0}$. Hence, $i_0 \le i$ and $s_{j_0} \equiv t_{j_0}$ for all j_0 with $1 \le j_0 < i_0$. For all k_0 with $i_0 < k_0 \le i$ we have $s \succ s_{k_0} \succcurlyeq t_{k_0}$. This implies by the case (γ) that $s \succ t$.

Both rules may compute n times a value for pair $\langle c, c' \rangle$ with costs bound by n, where n is the maximal arity of a function symbol. The overall costs of the rules are therefore similar to the costs of rules Monotonicity or Context. The running time of the test remains in $O(N^3 + |C|)$. □

With these additional rules for LPO it is possible to show more constraints unsatisfiable than without them.

Example 5.14 *Consider constraint $f(x, y) > f(y, x) \wedge y > x$. The result of preprocessing is $c_3 > c_4 \wedge c_2 > c_1$ with $\mathfrak{D}$ containing the rules $x \to c_1$, $y \to c_2$, $f(c_1, c_2) \to c_3$, and $f(c_2, c_1) \to c_4$. By* Ordering *we get $c_4 > c_2$ (i. e., $f(y, x) > y$), then by* Gamma *$c_4 > c_3$ (i. e., $f(y, x) > f(x, y)$) and by* Transitivity *$c_3 > c_3$, such that* Irreflexivity *applies. A test without the LPO-specific rules does not detect unsatisfiability, because there are ground reduction orderings for which the constraint is satisfiable (e. g., the RPOS with status right-to-left for function symbol f).*

Both rules for LPO work upwards using knowledge about the subterms. There are no rules for cases (α) and (δ) of the definition of LPO. The combination of rules Ordering and Transitivity covers these cases. We have thought about additional rules that work downwards. Unfortunately, such rules would introduce disjunctions, which we want to avoid to keep the worst-case runtime polynomial. Alternatively, we could restrict the downwards rules to situations where they would not introduce disjunctions. But this leads to rarely applicable rules with many conditions, which are expensive to check. Therfore, besides Beta and Gamma we do not use additional rules for LPO.

Additional rules for KBO

The KBO is parameterized by a weight function φ and a precedence $>_{\mathcal{F}}$. Handling the weight-based part of the ordering properly leads to the use of linear Diophantine equations and inequations (see Section 5.3). One of our main goals is to keep the test polynomial, and solving linear Diophantine equations is well-known to be NP-complete. Therefore, we use the weight-based part only for local tests in the following rules. Function ϕ is defined in Section 5.3 (see p. 92). It maps terms to linear sums taking into account φ. Let $\{x_1, \ldots, x_k\}$ be the variables of s and t, $\phi(s) = n_0 + \sum n_i x_i$ and $\phi(t) = m_0 + \sum m_i x_i$. We write $\phi(s) \geq \phi(t)$ if $n_i \geq m_i$ for $i = 1, \ldots, k$ and $n_0 + \sum n_i \mu \geq m_0 + \sum m_i \mu$, where μ is the weight of the smallest term. Therefore, $\phi(s) \geq \phi(t)$ implies $\varphi(\sigma(s)) \geq \varphi(\sigma(t))$ and $\phi(s) = \phi(t)$ implies $\varphi(\sigma(s)) = \varphi(\sigma(t))$ for any substitution σ.

1. KBO-DOWN:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c_i \geq c'_i\}, \mathfrak{D} \rangle$$

   if $\mathfrak{D}$ contains the rules $f(c_1, \ldots, c_n) \rightarrow c$ and $f(c'_1, \ldots, c'_n) \rightarrow c'$, $\mathfrak{C}$ contains $c \,\varrho\, c'$ with $\varrho \in \{>, \geq\}$, $1 \leq i < n$, $\mathfrak{C}$ contains $c_j = c'_j$ for all $j < i$, and $\phi(T_{\mathfrak{D}}(c)) = \phi(T_{\mathfrak{D}}(c'))$.

2. KBO-UP:

$$\langle \mathfrak{C}, \mathfrak{D} \rangle \quad \Longrightarrow \quad \langle \mathfrak{C} \cup \{c > c'\}, \mathfrak{D} \rangle$$

   if $\mathfrak{D}$ contains the rules $f(c_1, \ldots, c_n) \rightarrow c$ and $f(c'_1, \ldots, c'_n) \rightarrow c'$, there is some i with $1 \leq i < n$ such that $\mathfrak{C}$ contains $c_i > c'_i$ and $c_j = c'_j$ for all $j < i$, and $\phi(T_{\mathfrak{D}}(c)) \geq \phi(T_{\mathfrak{D}}(c'))$.

Note that for both rules case $i = n$ is excluded because it is already covered by rules CONTEXT and MONOTONICITY.

LEMMA 5.15 *Let $\succ$ be a ground-total KBO for precedence $>_{\mathcal{F}}$ and weight function φ. Let $\langle \mathfrak{C}, \mathfrak{D} \rangle \Longrightarrow \langle \mathfrak{C}', \mathfrak{D} \rangle$ with* KBO-DOWN *or* KBO-UP *and let $\sigma \in \mathrm{GSub}(\mathfrak{D})$. If σ satisfies $\langle \mathfrak{C}, \mathfrak{D} \rangle$ with respect to $\succ$, then σ satisfies $\langle \mathfrak{C}', \mathfrak{D} \rangle$ with respect to $\succ$. Extending the test by* KBO-DOWN *and* KBO-UP *does not increase its asymptotic running time.*

PROOF Let $s \equiv T_{\mathfrak{D}}(c)$ and $t \equiv T_{\mathfrak{D}}(c')$. Case KBO-DOWN: $\phi(s) = \phi(t)$ implies $\varphi(\sigma(s)) = \varphi(\sigma(t))$ for all ground substitutions σ. As s and t have the same top-symbol and $\sigma(T_{\mathfrak{D}}(c_j)) \equiv \sigma(T_{\mathfrak{D}}(c'_j))$ for all $j < i$, relation $\sigma(s) \succ \sigma(t)$ implies $\sigma(T_{\mathfrak{D}}(c_i)) \succeq \sigma(T_{\mathfrak{D}}(c'_i))$. Case KBO-UP: $\phi(s) \geq \phi(t)$ implies $\varphi(\sigma(s)) \geq \varphi(\sigma(t))$ for all ground substitutions σ. Hence, $\sigma(s) \succ \sigma(t)$ either by the weight or by the lexicographic comparison of the subterms.

Both rules may compute n times a value for pair $\langle c, c' \rangle$ with costs bound by n, where n is the maximal arity of a function symbol. The overall costs of the rules are therefore similar to the costs of rules MONOTONICITY or CONTEXT. The running time of the test remains in $O(N^3 + |C|)$. □

The two rules are somewhat complementary. For example, there are two ways to show the constraint considered in Example 5.14 unsatisfiable if $\succ$ is some KBO:

EXAMPLE 5.15 *Consider constraint* $f(x,y) > f(y,x) \wedge y > x$. *The result of preprocessing is* $c_3 > c_4 \wedge c_2 > c_1$ *with* $\mathfrak{D}$ *containing the rules* $x \rightarrow c_1$, $y \rightarrow c_2$, $f(c_1,c_2) \rightarrow c_3$, *and* $f(c_2,c_1) \rightarrow c_4$. *Using* KBO-DOWN, *from* $c_3 > c_4$ *(i. e.,* $f(x,y) > f(y,x)$*) we can derive* $c_1 \geq c_2$ *(i. e.,* $x \geq y$*), then* $c_2 > c_2$ *by* TRANSITIVITY, *hence* IRREFLEXIVITY *applies. Alternatively,* KBO-UP *allows us to derive* $c_4 > c_3$ *(i. e.,* $f(y,x) > f(x,y)$*) from* $c_2 > c_1$ *(i. e.,* $y > x$*), then* TRANSITIVITY *gives* $c_3 > c_3$, *and* IRREFLEXIVITY *applies as well.*

Similar to LPO, one can think about more elaborate rules for KBO. Especially the handling of the weight-based part could be improved. Some constraints are shown unsatisfiable very easily and quickly by the simplest test of Section 5.3. Very simple reasoning about the weights is sufficient. For these constraints, the generic test may fail to detect unsatisfiability even with the additional rules. For handling the weights, we could introduce an environment $\mathfrak{W}$ which captures lower and upper bounds of the weights for each subterm. If then the lower bound of $T_{\mathfrak{D}}(c)$ is greater than the upper bound of $T_{\mathfrak{D}}(c')$ we could deduce $c > c'$. Conversely, from $c \geq c'$ we could deduce that the weight of $T_{\mathfrak{D}}(c)$ is greater than or equal to the weight of $T_{\mathfrak{D}}(c')$. This suggests that $\mathfrak{W}$ should offer the facility to reason about transitive relations between weights: if the weight of s is greater than the weight of t and the weight of t is greater than or equal to the weight of u then the weight of s is greater than the weight of u. Taking into account the term structure even more complex relationships between the weights of terms can be deduced. It is unclear to us whether the weights and the associated bounds should be represented by linear sums initialized as $\phi(T_{\mathfrak{D}}(c))$ or whether simple numbers suffice. Using linear sums is of course more powerful, but may be very costly if solved over the naturals as we quickly reach NP-complete problems.

In more abstract terms, our wish is to find an improved handling of the weights that strengthens the test without deteriorating its performance. It should fit to the saturation approach of the algorithm and enable a bidirectional transfer of knowledge between the constraint part $\mathfrak{C}$ and the weight part $\mathfrak{W}$. This is a topic of future research.

5.5 Experimental evaluation

To evaluate the different tests for the unsatisfiability of ordering constraints we implemented them in the theorem prover WALDMEISTER [LH02]. We then performed test runs on machines equipped with Pentium III 1 GHz-processors and 4 GByte RAM. To decide linear arithmetic formulas we use the Omega-library, Version 1.2 [Pug92].

We are interested in two aspects of the tests. The first is the resource requirement of a test. It turns out that the memory requirements are modest for the different tests considered, so we concentrate on running times. The second aspect is the accuracy of the sufficient tests: How many of the unsatisfiable constraints are detected by the test? This is easy to determine for LPO as the method of Section 5.2 is a decision procedure. For KBO, however, we have no implementation of a decision procedure

available. As reference, we therefore use the union of all tests. This means that we consider a constraint as unsatisfiable if, and only if, it is detected unsatisfiable by at least one of the tests.

An important decision is the nature of the test sets we use for the evaluation. One choice is to use test sets consisting of randomly generated constraints. However, we do not think that this is appropriate. We can frequently observe that for NP-complete problems the instances occurring in practice are mostly well-behaved and exponential behavior is quite rare. We therefore decided to use test sets derived from our main domain of application for ordering constraints, the redundancy criteria that are topic of Chapter 6. As two of the criteria rely on the use of full ordering constraints, we use two groups of test sets. One group is derived from the ground joinability test based on confluence trees which is described in Section 6.4. The other group originates from the ground reducibility tests that are subject of Section 6.5.

For collecting the test sets we instrumented the prover to record for each invocation the arguments of the constraint satisfiability test. We then extracted the data from test runs that we used for evaluating the redundancy criteria. For the first group of test sets, we chose confluence tree variant $\textsc{ct}_5$. That specific implementation is described in Chapter 6 on p. 150. We selected the test runs with $\textsc{ct}_5$ that are depicted individually in Table 6.6 on p. 151 and do not suffer a timeout. We excluded two of the resulting test sets (BOO026-1 and GRP409-1) because they contain only few constraints. For the second group of test sets we chose the runs in column $\textsc{wm-acg}^+$ in Table 6.11 on p. 156 and used the constraints generated by the ground reducibility tests.

The test sets are then evaluated with $\succ$ being either an LPO or a KBO. As precedence we use the precedence used during the proof attempts. The weight function φ returns 1 for any symbol. The test sets contain instances that are trivial to detect unsatisfiable. These contain either an equality constraint that is not unifiable or an inequality constraint that can be shown unsatisfiable by a simple call to the ordering. Such trivial instances are omitted in the following evaluation.

Evaluation with respect to an LPO

For LPO we compare two different unsatisfiability tests: DP is an implementation of the decision procedure of Section 5.2; GT is an implementation of the generic test of Section 5.4. As can be seen in Table 5.1, most of the instances of the test sets belonging to group (a) are unsatisfiable, whereas most instances of the test sets belonging to group (b) are satisfiable.

Considering the running time of DP, we see that it is very fast in practice. An analysis of the backtrack-trees shows that branching occurs quite rarely and, because of the chosen heuristics h_{Sel}, mostly near the leaves. The redundancy elimination is powerful and unsatisfiable branches are usually short.

In comparison, GT on average takes about 6 times longer. The worst test set is RNG028-5 where GT takes about 20 times longer. In this test set, the average size of a constraint is by far the largest of all test sets, even after the preprocessing phase of the generic test, and GT performs on the average the most saturation steps per instance. This raises the question whether the generic test can be equipped with some kind

Table 5.1: Evaluation of accuracy and running times of two different methods to test LPO-constraints. Running times are in seconds.

test set	non-trivial	unsatis-fiable	DP		GT	
			accuracy	time	accuracy	time
(a) test sets derived from confluence trees						
BOO023-1	15 239	12 786	100.0%	0.36	98.5%	2.30
GRP181-3	60 853	50 884	100.0%	0.99	99.4%	5.31
LAT019-1	37 860	32 854	100.0%	1.45	99.3%	6.37
RNG028-5	26 168	17 456	100.0%	1.40	99.4%	27.71
ROB006-1	25 157	20 548	100.0%	0.38	98.5%	2.43
(b) test sets derived from ground reducibility tests						
LAT018-1	58 968	22 780	100.0%	3.97	98.7%	11.10
RNG036-7	134 005	33 311	100.0%	4.05	94.5%	13.65
ROB007-1	61 952	28 122	100.0%	1.39	99.9%	10.58
ROB020-1	27 372	11 024	100.0%	0.57	100.0%	4.36

of redundancy elimination to decrease the problem size, one of the strengths of the LPO-specific test. Concerning accuracy, GT is far better than expected. On average, it misses only 1.5 % of the unsatisfiable constraints.

Evaluation with respect to a KBO

For KBO we compare four different unsatisfiability tests: PHI, PHI_{H}, and PHI′ are implementations of the translation based tests of Section 5.3. For constraint C, they use the Omega-library to test the satisfiability of $\Phi(C) \wedge \text{Pos}(C)$, $\Phi_{\text{H}}(C) \wedge \text{Pos}(C)$, and $\Phi'(C) \wedge \text{Pos}(C)$. GT is an implementation of the generic test of Section 5.4. Table 5.2 presents the corresponding data.

Considering PHI and PHI_{H} we can see that PHI_{H} leads to a significantly stronger test for the test sets belonging to group (a). An analysis of the instances shows that most of them are related to the commutativity axiom C and its extension C'. Example 5.8 is therefore quite representative for the improvements of Φ_{H} over Φ. The application (confluence trees, see Section 6.4) often tests whether under a context of equality and ordering constraints an instance of C or C' has reducing ground instances. Whereas PHI does not decompose constraints of the form $s+t>t+s$ but adds the trivial inequation $0=0$ for these instances, PHI_{H} performs the decomposition and adds $\phi(s)-\phi(t)\geq 0$, which might be unsatisfiable together with the equations and inequations derived from the context. The application where group (b) of test sets originates is also concerned with reducing ground instances of C or C' (see Sections 6.5.1 and 6.5.2). However, here the application performs the decomposition, so PHI and PHI_{H} show no differences in accuracy for these test sets. This shows that some weaknesses of sufficient tests can be circumvented by the application.

Table 5.2: Evaluation of accuracy and running times of four different methods to test KBO-constraints. Running times are in seconds.

test set	non-trivial	unsatis-fiable	PHI		PHI_{H}		PHI'		GT	
			accuracy	time	accuracy	time	accuracy	time	accuracy	time
(a) test sets derived from confluence trees										
BOO023-1	5 825	2 044	29.5%	0.95	51.2%	0.80	52.0%	1.57	98.6%	1.10
GRP181-3	51 027	37 735	2.3%	7.98	28.1%	7.25	29.1%	18.56	99.7%	7.14
LAT019-1	32 050	27 446	2.0%	6.40	40.3%	6.55	40.8%	13.85	99.9%	8.52
RNG028-5	10 664	6 191	7.2%	1.83	30.6%	1.57	45.6%	4.82	95.9%	12.43
ROB006-1	22 249	18 133	0.3%	3.41	32.6%	2.98	33.7%	7.56	99.9%	3.16
(b) test sets derived from ground reducibility tests										
LAT018-1	59 789	21 402	54.5%	17.18	54.5%	14.35	55.8%	39.10	97.8%	14.58
RNG036-7	134 141	34 220	38.5%	26.19	38.5%	21.10	44.6%	47.11	86.7%	16.11
ROB007-1	65 102	28 766	77.0%	9.87	77.0%	7.96	77.9%	14.77	98.9%	8.44
ROB020-1	27 909	11 282	67.1%	4.28	67.1%	3.44	68.2%	6.04	98.7%	3.30

The test PHI' is stronger than PHI_{H}, which indicates that the translation is more accurate. However, PHI' requires more than twice the time of PHI_{H}. This seems justified as the costs of the test are on the average still well below one millisecond per invocation. However, one should be very careful introducing disjunctions. Our first version of PHI' replaced $s \geq t$ by $s = t \vee s > t$ for all constraints. This test required more than 10 000 seconds for the test set LAT018-1!

In comparison, GT is quite well-behaved. Except for RNG028-5 (see the previous discussion), is requires roughly the same time as PHI_{H}. However, GT has a far better accuracy which is on the average at 96.8 %. Analyzing the instances where it fails to detect unsatisfiability shows that typically simple conclusions about the weights suffice to do so. Two patterns occur frequently: The first concerns transitive conclusions about the weights. For example, if w_1 is greater than w_2 which is itself greater than w_3 and w_3 is greater than w_1, then there is no solution as by transitivity w_1 should be greater than itself. The second pattern has the following form: Weight w_1 is greater than w_2 which is greater than w_3 which is greater than w_4. However, the difference $w_2 - w_3$ is greater than the difference $w_1 - w_4$. Hence, no solution can exist. This shows that there is room for improvements for adapting the generic test to KBO. Some ideas are outlined at the end of Section 5.4.

5.6 Conclusions

Ordering and equality constraints offer a convenient and powerful formalism to restrict in a precise way the (ground) instances of a syntactic expression. The main algorithmic problem is to check whether a given constraint is satisfiable or not. We can distinguish whether a method for that test is a decision procedure or only a sufficient test and

whether it is ordering specific or generic in nature. We presented five different methods: a decision procedure for LPO constraints in Section 5.2, three related sufficient tests for KBO constraints in Section 5.3, and a generic test in Section 5.4.

The experiments of Section 5.5 show that ordering constraints can be tested for satisfiability in reasonable time – at least for the test sets that we have considered. This makes the use of constraints feasible in high-performance systems. However, one should relate the costs of using constraints to the costs of other operations such as simplification. For example, in our experience constraint-based critical-pair criteria are still too expensive to justify their use.

Of the five tests evaluated, the generic test GT shows (despite its genericity!) a very good accuracy at moderate costs. For KBO, it is clearly the best choice. Of the four considered tests it has by far the best accuracy at reasonable running times. The translation based methods might be easier to implement, provided a library to decide linear arithmetic is available. However, one is then dependent on the peculiarities of the library. For LPO, the decision procedure DP is faster than the generic test GT and is not only an approximation. However, in our experience with DP and GT, the implementation of the generic test is much simpler than the implementation of the decision procedure. We had to invest a considerable development effort to make DP as fast as it is now. With all the improvements described in Section 5.2 it is for simple constraints about twenty times faster than the original implementation used for the experiments of [NR02]. For more complex constraints (e. g. from the test set RNG028-5), we observe differences of three to more than five orders of magnitude. In such cases the developed heuristics show their strength.

If one needs an unsatisfiability test for ordering constraints and it is not strictly necessary to use a decision procedure, we strongly recommend implementing the generic test. It has a very good balance between implementation complexity, runtime efficiency, and detection accuracy. Much work in the literature has been devoted to the development of ordering-specific decision procedures (e. g. [Com90, Nie93b, NRV98, KV01, NR02]). Nieuwenhuis posed the question whether there exists efficient (but only sufficient) tests for detecting most of the constraints unsatisfiable [Nie99, open problem 7]. In our opinion, the generic test of Section 5.4 (first presented in [Löc04a]) gives a first solid answer to this question.

6 Redundancy criteria

In this chapter we present and evaluate several redundancy criteria, which are all based on ground joinability. As developed in Chapter 3, redundancy criteria are an important means to strengthen the usual redundancy elimination mechanism, namely interreduction. For their evaluation we need more than a simple comparison of detection strength and computational cost. Hence, in Section 6.1 we discuss various aspects of redundancy criteria that determine their usability in practice. We continue in Section 6.2 with a simple, yet powerful approach which uses ground convergent subsystems. The next two sections are devoted to techniques based on case splitting which is described by constraints. They differ in the form of constraints used: The criterion of Section 6.3 uses simple variable constraints, the criterion of Section 6.4 uses full ordering constraints. Whereas the methods mentioned so far only declare an equation as redundant when it is already ground joinable, the technique of Section 6.5 has a different approach: It detects equations that are ground reducible and *makes* them ground joinable by adding the necessary overlaps. We proceed with an experimental evaluation of the different criteria in Section 6.6. We conclude in Section 6.7.

In order to simplify the presentation we omit in this chapter the distinction between the sets E and G and assume that all operations are well-sorted.

6.1 Practical aspects of redundancy criteria

For the evaluation of redundancy criteria it is not sufficient to consider only the detection strength. From a practical viewpoint the cost-benefit ratio, i. e., the relation of required running time to detection strength, is more interesting. The computational requirements of a test should not outweigh its potential benefits. An asymptotic complexity analysis is often difficult because a test may require normalization steps. These steps are inherently dependent on the actual rewrite system and therefore in most cases elude a precise theoretical analysis. So we have to rely on experiments for the evaluation. This raises another important aspect: the difficulty of implementation.

Some knowledge about the implementation of the completion process is necessary for the assessment of a redundancy criterion. (A more detailed account of this prover architecture is topic of Chapter 7.) The search state is mainly determined by two sets, the set $\mathfrak{A}$ of *active facts* and the set $\mathfrak{P}$ of *passive facts*. One iteration in the completion process consists of the following steps: (1) Select an element of $\mathfrak{P}$ by some heuristics φ and normalize it with the elements of $\mathfrak{A}$. (2) If not joinable, use the new equation to interreduce the elements of $\mathfrak{A}$ and then add it to $\mathfrak{A}$ (i. e., *activate* it). (3) Compute all critical pairs between the new equation and the elements of $\mathfrak{A}$, normalize them, and insert the nontrivial ones into $\mathfrak{P}$.

Note that most of the work is induced by step (3). Interreduction is beneficial because an equation does not only contribute at the time of its activation to the generation of critical pairs. Later in the completion process, it serves as second equation when the critical pairs of the then activated equation are computed. Furthermore, critical pairs of the interreduced equation that have not yet been activated can be deleted.

Traditionally, the rewrite steps in (1) and (3) are called *forward simplification*, whereas the rewrite steps in (2) are called *backward simplification*. For a redundancy criterion, which is more costly than normalization, it is only sensible to enrich steps (1) and (2). In step (3) its use would merely allow us to save some space by deleting additional critical pairs. However, for this purpose specialized compression techniques are more appropriate (see Chapter 7). The main purpose of the rewrite steps in (3) is to improve the heuristic evaluation of the critical pairs.

For a redundancy criterion we can therefore distinguish whether it is sufficient to use it as a forward test in step (1), or whether it is necessary to perform backward tests in step (2). The latter has the benefit to take into account later developments of the completion process and therefore to detect more equations redundant. However, the use as backwards test is also some kind of weakness, which might be surprising at first sight. There are two effects to consider.

First, a successful redundancy test has the benefit that the prover can avoid the computation of critical pairs with the equation at hand. If a forward test succeeds, then it computes no critical pairs with the equation. If a backward test succeeds, it can avoid only the part of the critical pairs that would afterwards be computed. The initial set of critical pairs is still computed at the activation of the equation in step (3). In hindsight, this turns out to be superfluous for all or most of them. A typical pattern is that a backwards test is successful right after the activation of the next or the next few equations. Often, a critical pair of the equation can be used to show the equation redundant.

A second issue with backward tests is the problem to perform them efficiently when the size of $\mathfrak{A}$ is in the order of several hundreds or thousands equations. This is of great importance, because the backward test is performed once per iteration cycle on all (or most) elements of $\mathfrak{A}$. In contrast, a forward test is performed only once per equation, which allows it to be more elaborate.

Another important aspect is the question whether we should delete a redundant equation or keep it for simplification. Inference system $\mathcal{G}$ offers both options. An alternative may be to replace redundant equations by a set of (redundant) equations that better fit the purpose of simplification. A further approach is to make an equation redundant by explicitly adding a set of simpler, nonredundant equations. Here, too, we have to rely on experiments, because an equation that is not necessary for the proof may nevertheless be important for the proof search.

Finally, we may not neglect questions that have a more theoretic flavor. What are the limitations of a criterion? Is it fixed to a specific theory or generic in nature? What are the parameters of its complexity? How do they limit its applicability? Are there efficient approximations?

6.2 Ground convergent subsystems

The first criterion we present is based on subsystems of R and E that are ground convergent[1]. Such subsystems frequently occur in practice. As the automatic detection of such systems seems not feasible, we rely on a catalog of well-known ground convergent systems. The whole approach is based on the following theorem.

THEOREM 6.1 *Let $R_0 \subseteq R$ and $E_0 \subseteq E$ with $R_0(E_0)$ ground convergent. Consider equation $s = t$ with $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. If $s =_{R_0 \cup E_0} t$ then $s \Downarrow t$ in $R_0(E_0)$. If furthermore $s = t \notin R_0 \cup E_0$, $s = t$ is not "distinguished", and all equations in E are "distinguished", then $s \Downarrow_{\triangleright} t$ in $R_0(E_0)$, hence $s = t \succ_{\mathcal{P}} R(E)$.*

PROOF If $s =_{R_0 \cup E_0} t$ we have $\sigma(s) =_{R_0 \cup E_0} \sigma(t)$ for all $\sigma \in \mathrm{GSub}(s, t)$. By ground convergence of $R_0(E_0)$ it follows that then $\sigma(s) \downarrow \sigma(t)$ in $R_0(E_0)$, which implies $s \Downarrow t$ in $R_0(E_0)$. The additional condition for $s \Downarrow_{\triangleright} t$ requires for each instance $\sigma(s) = \sigma(t)$ that the first rewrite step on the maximal side is either performed at a position $p \neq \lambda$ or with an equation $l = r$ such that $(s, t) \triangleright (l, r)$. This condition is easily met in a forward test. As $s = t$ is then a passive equation, the existence of a match from $l = r$ to $s = t$ implies $(s, t) \triangleright (l, r)$. To guarantee analogously the condition for a backward test, however, the elements of E_0 have to be "distinguished" equations – in contrast to $s = t$. As $R_0 \subseteq R$ and $E_0 \subseteq E$ we have by Theorem 3.3 that $s = t \succ_{\mathcal{P}} R(E)$. □

Making the elements of E_0 "distinguished" is a reasonable decision: It enables more redundancy proofs and avoids the use of explicit $\triangleright$-tests.

The test for confluence of a terminating rewrite system can be reduced to the joinability of critical pairs. This is a consequence of Newman's lemma [New42] in combination with a careful divergence analysis together with the critical-pair lemma [KB70]. A similar result holds for the test for ground confluence of an ordered rewrite system. This is essentially a result of the unfailing completion approach (see [HR87], and also [BDH86, BDP89]), but was explicitly formulated only later in [MN90, CNNR03].

LEMMA 6.1 *An ordered rewrite system $R_0(E_0)$ is ground convergent iff $s \Downarrow t$ in $R_0(E_0)$ for all $s = t$ in $\mathrm{CP}(R_0, E_0)$.* □

Many specifications contain AC-axioms for an operator. The system ACC' (see Chapter 3.2, p. 15) is ground convergent:

EXAMPLE 6.1 *If $\succ$ is an LPO or a KBO total on ground terms then the system with $R_0 = \{(x + y) + z \rightarrow x + (y + z)\}$ and $E_0 = \{x + y = y + x, x + (y + z) = y + (x + z)\}$ is ground convergent. The methods of Chapter 6.3 and Chapter 6.4 show this automatically. Here we want to give an intuitive explanation. For any critical pair $s = t$ in $\mathrm{CP}(R_0, E_0)$ the R_0-normal form is either $x_1 + (x_2 + x_3) = x_{\pi(1)} + (x_{\pi(2)} + x_{\pi(3)})$ or $x_1 + (x_2 + (x_3 + x_4)) = x_{\pi(1)} + (x_{\pi(2)} + (x_{\pi(3)} + x_{\pi(4)}))$, π some permutation. For*

[1] The systems should be ground convergent, but not convergent. The main redundancy elimination mechanism, simplification, already exploits the advantages of convergent subsystems.

any ground instance of these normal forms, described by $\sigma \in \mathrm{GSub}(s,t)$*, the equations in* E_0 *allow us to sort the AC-subterms of both sides in increasing order – simulating bubble-sort. As both sides contain the same AC-subterms and* $\succ$ *is total on ground terms, this implies* $\sigma(s) \downarrow \sigma(t)$*. Therefore, all critical pairs are ground joinable in* $R_0(E_0)$*.*

Note that for showing $(\mathsf{M})\ x + (y + z) = z + (y + x)$ and $(\mathsf{S})\ x + (y + z) = y + (z + x)$ redundant it is essential to make equations C and C' "distinguished". Otherwise, we would depend on the activation order of equations.

An extension of ACC' is important for example in the domain of lattices.

Example 6.2 *If for an operator additionally to AC the idempotence axiom is present,* R_0 *of the previous example can be extended by* $x + x \to x$ *and* $x + (x + y) \to x + y$ *and* E_0 *can be kept the same (cf. Example 3.3, p. 16). The resulting system* $R_0(E_0)$ *is ground convergent, which can easily be seen extending the intuitive argumentation: After sorting the AC-subterms the new rules allow us to eliminate duplicates, which are located side by side, as* $\succ$ *is total on ground terms.*

In some situations the combination of ground convergent subsystems is ground convergent as well. In the next lemma we consider a sufficient condition.

Lemma 6.2 *Let* $R_1(E_1)$ *and* $R_2(E_2)$ *be two ground convergent subsystems of* $R(E)$ *and let* $R_{12} = R_1 \cup R_2$ *and* $E_{12} = E_1 \cup E_2$*. If there are no critical pairs between elements of* $R_1 \cup E_1$ *and elements of* $R_2 \cup E_2$ *then* $R_{12}(E_{12})$ *is ground convergent.*

Proof This is mainly a consequence of Lemma 6.1. We partition the set of critical pairs $CP = \mathrm{CP}(R_{12}, E_{12})$ into three disjunct subsets: $CP_1 = \mathrm{CP}(R_1, E_1)$, $CP_2 = \mathrm{CP}(R_2, E_2)$, and $CP' = CP - (CP_1 \cup CP_2)$. For $i = 1, 2$ the elements of CP_i are ground joinable in $R_i(E_i)$ as this subsystem is ground convergent. This implies that they are ground joinable in $R_{12}(E_{12})$. Furthermore $CP' = \varnothing$ by assumption. Therefore, all critical pairs are ground joinable and thus $R_{12}(E_{12})$ is ground convergent. □

This is useful in the domain of lattices or in the domain of commutative rings, where two AC-operators are present.

Example 6.3 *Consider a specification with several AC-operators (i. e.,* $|\mathcal{F}\ \ | > 1$*). Let* R_0 *contain for each AC-operator the associativity axiom* A *oriented as rule and let* E_0 *contain for each AC-operator the commutativity axiom* C *and its extension* C'*. Then* $R_0(E_0)$ *is a ground convergent rewrite system.*

Implementation aspects

Considering the implementation of this criterion, two approaches come into mind. The first is the standard rewriting approach. As $R_0(E_0)$ is ground convergent and ground convergence is stable against signature extensions, the word problem $s =_{R_0 \cup E_0} t$ is decidable by Skolemizing s and t and then performing a single joinability test (cf. Proposition 3.1).

Proposition 6.1 *Let* $R_0(E_0)$ *be ground convergent and* $s = t$ *be the equation at hand. Let* $\bar{s} = \bar{t}$ *be the Skolemized version of* $s = t$*, that is, each variable* $x_i \in \mathrm{Var}(s,t)$ *is replaced by a (fresh) Skolem constant* c_i*. Then* $s =_{R_0 \cup E_0} t$ *iff* $\bar{s} \downarrow \bar{t}$ *in* $R_0(E_0)$*.*

Proof If there is a proof P for $\bar{s} = \bar{t}$ in $R_0(E_0)$ then there is also a rewrite proof P' for $\bar{s} = \bar{t}$ in $R_0(E_0)$: any peak $u \longleftarrow_{R_0(E_0)} v \longrightarrow_{R_0(E_0)} w$ can be replaced by a smaller sub-proof $u \xrightarrow{*}_{R_0(E_0)} v' \xleftarrow{*}_{R_0(E_0)} w$ because $R_0(E_0)$ is ground confluent. □

This approach is not only general but also quite efficient: For one equation we only have to normalize two terms after Skolemization. In the case of AC-operators the number of rewrite steps performed is in the worst case quadratic in the size of the terms as the rewriting process simulates some kind of bubble-sort. This observation suggests a second approach to implement this criterion: the use of decision procedures to determine whether $s =_{R_0 \cup E_0} t$ holds. In the case of AC-operators we can directly use sorting:

Theorem 6.2 *Let* $R_0(E_0)$ *be the ground convergent rewrite system for one or more AC-operators (as in Example 6.3) and let* $s = t$ *be the equation at hand. Let* $>$ *be a total ordering on* $\mathrm{Term}(\mathcal{F}, \mathcal{V})$*. Let* $\tilde{s} = \tilde{t}$ *result from* $s = t$ *by the following transformation: We perform an innermost traversal on the* R_0*-normal forms and replace maximal subterms of the form* $t_1 + \ldots + t_n$*,* $+ \in \mathcal{F}$ *, by* $t_{\pi(1)} + \ldots + t_{\pi(n)}$*,* π *some permutation, such that the sequence* $t_{\pi(1)}, \ldots, t_{\pi(n)}$ *is sorted with respect to* $>$*. Then* $s =_{R_0 \cup E_0} t$ *iff* $\tilde{s} \equiv \tilde{t}$*.*

Proof The “if”-direction is straightforward, as any replacement of $t_1 + \ldots + t_n$ by $t_{\pi(1)} + \ldots + t_{\pi(n)}$ with π some permutation can be achieved by a finite number of E_0-steps. For the “only if”-direction assume the contrary and consider some minimal counter example $s = t$ for which we have $s =_{R_0 \cup E_0} t$ and $\tilde{s} \not\equiv \tilde{t}$. As s and t are minimal $\tilde{s} \equiv s_1 + \ldots + s_n$ and $\tilde{t} \equiv t_1 + \ldots + t_m$, with $+ \in \mathcal{F}$. Because rewrite steps with $R_0 \cup E_0$ as well as the transformation are length preserving we have $n = m$. Furthermore, there exists a bijection π such that $s_i =_{R_0 \cup E_0} t_{\pi(i)}$ and, as s and t are minimal counter examples, $s_i \equiv t_{\pi(i)}$. But then the sorted sequences of $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ are identical. □

Note that in the proof we rely on the use of an innermost traversal. As the only requirement on $>$ is to be a total ordering on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ the transformation may change the ordering relation between two subterms. An obvious choice for $>$ is to consider the terms as words and to use the lexicographic extension of a total ordering on $\mathcal{F} \cup \mathcal{V}$. Because we can use an efficient sorting procedure in the transformation the complexity of this implementation is in $O(k \log k)$, $k = |s| + |t|$. The generalization of this method to ACI-operators is straightforward: We eliminate identical subterms during sorting.

A rough comparison of the rewriting based method with the sorting based method shows the latter to be typically about three times faster. As the time needed for this criterion is only a fraction of one per cent of the overall run-time, one may nevertheless choose the rewriting method for its greater flexibility.

One big advantage of the criterion is that we can use it in a purely forward way. In the common case it is sufficient to analyze the input specification and to search for a known subset of equations, such as AC or ACI. As we have to protect the elements of R_0 and E_0 anyway, we can simply start filtering the equations from the beginning of the completion. Missing elements of R_0 and E_0, such as in the case of AC the extension of the commutativity axiom C', are then generated by the completion process. Alternatively, we can enrich the specification by such missing equations in a preprocessing step. This does not change the theory, as these equations are logical consequences of the specification.

Sometimes the specification does not contain any known ground convergent subset. Then we can monitor during the completion whether the elements of some subset get activated. We start the filtering when we find a known ground convergent subset. At this point it seems advantageous to once perform a backward test, which amounts to a simple test for each element of $\mathfrak{A}$. Other backward tests are not necessary.

6.3 Case splitting with variable constraints

This criterion allows us to go beyond fixed theories. It is based on splitting by cases, which are described by constraints on the variables. Suppose $s = t$ is the equation at hand and $\mathrm{Var}(s,t) = \{x_1, \ldots, x_n\}$. In [MN90] it is proposed to consider any ordering relation $\sigma(x_{i_1}) \sim_1 \ldots \sim_{n-1} \sigma(x_{i_n})$, $\sim_i \in \{\succ, \equiv\}$, between ground instances $\sigma(x_1), \ldots, \sigma(x_n)$ and to prove $s \downarrow t$ under each of these constraints. This implies $s \Downarrow t$; the refinement needed for $s \Downarrow_{\rhd} t$ is straightforward: For every instance to be analyzed, we just have to be careful with the first rewrite step on the maximal side of $\sigma(s) = \sigma(t)$. If it occurs top-level we have to perform an $\rhd$-test explicitly.

The number of cases grows exponentially with n; starting with $n = 0$ we have $1, 1, 3, 13, 75, 541, 4683, \ldots$ cases (see [Slo05] for more information about this sequence). To be useful in practice, we have to limit the number of cases. As a rule of thumb, their number should not exceed the number of critical pairs that are immediately computed when the equation at hand gets activated. A typical value is at most a few hundred cases, in some situations perhaps some five hundred cases may be justified.

Therefore we have to refine the idea of [MN90]. Instead of doing case splits with respect to all variables occurring in $s = t$, we consider only a subset for splitting and try to show ground joinability with this restricted split. If this is not successful we can extend the subset or choose a different one. Recall that this is a sufficient test only, which is used to speed up the proof search. So it is advantageous if an equation can be shown redundant, but it is unproblematic if one remains undetected.

To make the approach precise we model the ordering constraints on variable instances by a quasi-ordering $\succsim_V$ on $V_0 \subseteq \mathrm{Var}(s,t)$ and extend the reduction ordering $\succ$ to a quasi-ordering $\succsim_0$ on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$ containing $\succsim_V$. We first formulate requirements on $\succsim_0$ that will allow us to approximate rewriting on $\mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ by rewriting on $\mathrm{Term}(\mathcal{F}, \mathcal{V})$.

DEFINITION 6.1 *A quasi-ordering* $\succsim_0$ covers *a ground substitution* $\sigma \in \mathrm{GSub}(s,t)$ *with respect to a reduction ordering* $\succ$ *if for* $x, y \in \mathcal{V}$ *the relation* $x \succ_0 y$ *implies* $\sigma(x) \succ \sigma(y)$ *and the relation* $x \approx_0 y$ *implies* $\sigma(x) \equiv \sigma(y)$.

The ordering $\succsim_0$ *on* $\mathrm{Term}(\mathcal{F},\mathcal{V})$ *is* compatible *with* $\succ$ *if we have for all* $s,t \in \mathrm{Term}(\mathcal{F},\mathcal{V})$ *and all* $\sigma \in \mathrm{GSub}(s,t)$ *covered by* $\succsim_0$ *that* $s \succ_0 t$ *implies* $\sigma(s) \succ \sigma(t)$.

EXAMPLE 6.4 *The quasi-ordering* $x \succsim_0 y$ *covers the substitution* $\sigma = \{x \mapsto f(a), y \mapsto a\}$ *with respect to an arbitrary LPO* $\succ$. *We then have for example* $s \equiv x+y \succsim_0 y+x \equiv t$, *which corresponds to* $\sigma(s) \equiv f(a) + a \succ a + f(a) \equiv \sigma(t)$. *Later we will show for any LPO* $\succ$ *how to extend it with* $\succsim_V$ *such that the extended ordering* $\succsim_0$ *is compatible with* $\succ$.

In case of compatibility, rewrite steps under $\succ_0$ (written as $s \longrightarrow^{\succ_0}_{R(E)} t$) imply rewrite steps under $\succ$ on the covered ground instances:

LEMMA 6.3 *Let* $\succsim_0$ *be compatible with* $\succ$. *If* $s \longrightarrow^{\succ_0}_{R(E)} t$, *then* $\sigma(s) \longrightarrow_{R(E)} \sigma(t)$ *for all* $\sigma \in \mathrm{GSub}(s,t)$ *that are covered by* $\succsim_0$.

PROOF Let $s \longrightarrow^{\succ_0}_{R(E)} t$ with $l = r \in R \cup E$, substitution θ and position $p \in \mathcal{O}(s)$, that is, $s \equiv s[\theta(l)]_p \longrightarrow s[\theta(r)]_p \equiv t$ and $\theta(l) \succ_0 \theta(r)$. As $\succsim_0$ is compatible with $\succ$, we have for any covered $\sigma \in \mathrm{GSub}(s,t)$ that $\sigma(\theta(l)) \succ \sigma(\theta(r))$ and therefore $\sigma(s) \equiv \sigma(s)[\sigma(\theta(l))]_p \longrightarrow_{R(E)} \sigma(s)[\sigma(\theta(r))]_p \equiv \sigma(t)$. □

EXAMPLE 6.5 *Continuing the previous example, under the quasi-ordering* $x \succsim_0 y$ *the commutativity axiom allows us to rewrite the term* $g(x+y)$ *to* $g(y+x)$. *This covers the rewrite step* $g(f(a)+a) \longrightarrow g(a+f(a))$ *of the ground instance described by* σ.

Iterating the argumentation, we get the same result for joinability under $\succ_0$ (written as $\downarrow^{\succ_0}$):

COROLLARY 6.1 *Let* $\succsim_0$ *be compatible with* $\succ$. *If* $s \downarrow^{\succ_0} t$, *then* $\sigma(s) \downarrow \sigma(t)$ *for all* $\sigma \in \mathrm{GSub}(s,t)$ *that are covered by* $\succsim_0$. □

This enables us to formulate the following test for ground joinability.

THEOREM 6.3 *Let* $\succ$ *be a reduction ordering and* $s,t \in \mathrm{Term}(\mathcal{F},\mathcal{V})$ *be two terms. Let* $O = \{\succsim_1, \ldots, \succsim_k\}$ *be a set of term orderings such that for each* $\succsim_i \in O$ *the ordering* $\succsim_i$ *is compatible with* $\succ$ *and that furthermore for each* $\sigma \in \mathrm{GSub}(s,t)$ *there exists an ordering* $\succsim_i \in O$ *that covers* σ.

If we have for all $i \in \{1,\ldots,k\}$ *that* $s \downarrow^{\succ_i} t$, *then* $s \Downarrow t$.

PROOF We have to show for each $\sigma \in \mathrm{GSub}(s,t)$ that $\sigma(s) \downarrow \sigma(t)$ holds. By assumption, there exists an ordering $\succsim_i \in O$ that covers σ. Because $\succsim_i$ is compatible with $\succ$ and further $s \downarrow^{\succ_i} t$ we have $\sigma(s) \downarrow \sigma(t)$ by Corollary 6.1. Therefore, $s \Downarrow t$. □

Two questions remain: (1) Given a reduction ordering $\succ$ and a constraint $\succsim_V$ on V_0, how to find an ordering $\succsim_0$ such that $\succsim_V \subseteq \succsim_0$ and $\succsim_0$ is compatible with $\succ$? (2) How to find a small set O of orderings that cover each $\sigma \in \mathrm{GSub}(s,t)$?

Compatible orderings for LPO and KBO

We answer question (1) for the LPO and the KBO by extending their definitions to take into account $\succsim_0$. This differs from the approach of [MN90]. There an abstract closure operator $\mathcal{C}(\cdot)$ is used to capture the necessary properties of the ordering. In their Prolog-implementation $\mathcal{C}(\cdot)$ is then approximated by additional clauses specific to the example.

First, we consider the extension of syntactic identity by quasi-ordering $\succsim_V$.

LEMMA 6.4 *Let $\succsim_V$ be a quasi-ordering on V_0. Let $\approx_0$ be given by*

- *$x \approx_0 y$ iff $x \approx_V y$ for $x, y \in \mathcal{V}$*
- *$f(s_1, \dots, s_n) \approx_0 f(t_1, \dots, t_n)$ iff $s_1 \approx_0 t_1, \dots, s_n \approx_0 t_n$.*

Let $\succsim_V$ cover σ with respect to some ordering $\succ$. Then $s \approx_0 t$ implies $\sigma(s) \equiv \sigma(t)$.

PROOF Induction on $|s| + |t|$. If $x \approx_0 y$ with $x, y \in \mathcal{V}$ we have by definition $x \approx_V y$ and as $\succsim_V$ covers σ this implies $\sigma(x) \equiv \sigma(y)$. If $s \approx_0 t$ with $s, t \notin \mathcal{V}$ we have by definition $s \equiv f(s_1, \dots, s_n)$, $t \equiv f(t_1, \dots, t_n)$, and $s_1 \approx_0 t_1, \dots, s_n \approx_0 t_n$. By induction hypothesis we get $\sigma(s_1) \equiv \sigma(t_1), \dots, \sigma(s_n) \equiv \sigma(t_n)$, hence $\sigma(s) \equiv \sigma(t)$. □

For LPO the intuitive background for the following construction of the extended ordering is to consider for $\succsim_0$ variables as constants and to use then an LPO with quasi-precedence $>_{\mathcal{F}} \cup \succsim_V$.

LEMMA 6.5 *Let $\succ$ be the LPO with precedence $>_{\mathcal{F}}$ and let $\succsim_V$ be a quasi-ordering on V_0. Let $\approx_0$ as in Lemma 6.4, $\succsim_0 = \succ_0 \cup \approx_0$, and let $\succ_0$ be given by: $s \succ_0 t$ iff either $s \equiv f(s_1, \dots, s_n)$, $t \equiv g(t_1, \dots, t_m)$, and*

- (α) *$s_i \succsim_0 t$ for some $i \in \{1, \dots, n\}$ or*
- (β) *$f >_{\mathcal{F}} g$ and $s \succ_0 t_k$ for all $k \in \{1, \dots, m\}$ or*
- (γ) *$f = g$, there exists some $i \in \{1, \dots, m\}$ such that $s_j \approx_0 t_j$ for all $j \in \{1, \dots, i-1\}$ and $s_i \succ_0 t_i$, and $s \succ_0 t_k$ for all $k \in \{1, \dots, m\}$*

or $s \equiv f(s_1, \dots, s_n)$, $t \equiv x$, $x \in \mathcal{V}$ and

- (δ) *$s_i \succsim_0 x$ for some $i \in \{1, \dots, n\}$,*

or $s \equiv x$, $t \equiv y$, $x, y \in \mathcal{V}$ and

- (ε) *$x \succ_V y$.*

Then $\succsim_0$ is compatible with $\succ$.

PROOF Let $\succsim_0$ cover $\sigma \in \mathrm{GSub}(s, t)$ with respect to $\succ$. We show by induction on $|s| + |t|$ that then $s \succ_0 t$ implies $\sigma(s) \succ \sigma(t)$ inspecting the possible justifications for $s \succ_0 t$.

- (α) $s \equiv f(s_1, \dots, s_n)$ and $s_i \succsim_0 t$ for some i. If $s_i \approx_0 t$, then $\sigma(s_i) \equiv \sigma(t)$ by Lemma 6.4. Otherwise, we have $s_i \succ_0 t$ and by induction hypothesis $\sigma(s_i) \succ \sigma(t)$. In both cases $\sigma(s_i) \succcurlyeq \sigma(t)$ holds, hence $\sigma(s) \succ \sigma(t)$.

(β) $s \equiv f(s_1, \ldots, s_n)$, $t \equiv g(t_1, \ldots, t_m)$, $f >_{\mathcal{F}} g$, and $s \succ_0 t_k$ for all $k \in \{1, \ldots, m\}$. The induction hypothesis implies $\sigma(s) \succ \sigma(t_k)$ for all k and thus $\sigma(s) \succ \sigma(t)$.

(γ) $s \equiv f(s_1, \ldots, s_m)$, $t \equiv f(t_1, \ldots, t_m)$, there exists some $i \in \{1, \ldots, m\}$ such that $s_j \approx_0 t_j$ for all $j \in \{1, \ldots, i-1\}$ and $s_i \succ_0 t_i$, and $s \succ_0 t_k$ for all $k \in \{1, \ldots, m\}$. Again $s_j \approx_0 t_j$ implies $\sigma(s_j) \equiv \sigma(t_j)$ for $j < i$ by Lemma 6.4. Furthermore, by induction hypothesis we have $\sigma(s_i) \succ \sigma(t_i)$ and $\sigma(s) \succ \sigma(t_k)$ for all k. This implies $\sigma(s) \succ \sigma(t)$.

(δ) Identical to case (α).

(ε) $s \equiv x$, $t \equiv y$, $x, y \in \mathcal{V}$, and $x \succ_V y$. By construction, $\succsim_0$ and $\succsim_V$ coincide on $\mathcal{V}$. As $\succsim_0$ covers σ with respect to $\succ$, we have $\sigma(x) \succ \sigma(y)$.

□

Note that the ordering $\succsim_0$ coincides with $\succcurlyeq$, the standard LPO for $>_{\mathcal{F}}$, if the variable constraint is minimal (i. e., $\succsim_V = \equiv$).

Example 6.6 *Let $s \equiv f(x, a)$, $t \equiv f(y, y)$, and $f >_{\mathcal{F}} a$. Under the constraint $x \succ_V y$ we have $s \succ_0 t$, as $f = f$, $x \succ_0 y$, and $f(x, a) \succ_0 y$ because of $x \succsim_0 y$. Under the constraint $y \succ_V x$ we have $t \succ_0 s$, as $f = f$, $y \succ_0 x$, and $f(y, y) \succ_0 a$ as $f >_{\mathcal{F}} a$. Under the constraint $x \approx_V y$, however, s and t are not comparable by $\succsim_0$: As $f = f$ and $x \approx_0 y$ the ordering relation between s and t is determined by the ordering relation between a and y – which are incomparable.*

For the KBO a result similar to Lemma 6.5 holds. Recall that $|t|_x$ denotes the x-length of t, that is, the number of occurrences of x in t.

Lemma 6.6 *Let $>_{\mathcal{F}}$ be a precedence on $\mathcal{F}$ and $\succsim_V$ a quasi-ordering on V_0. Let $\varphi : \mathcal{F} \cup \mathcal{V} \to \mathbb{N}$ be a weight function admissible to $>_{\mathcal{F}}$ with $\varphi(x) = 1$ for all $x \in \mathcal{V}$. Let $\approx_0$ as in Lemma 6.4, $\succsim_0 = \succ_0 \cup \approx_0$, and let $\succ_0$ be given by:*

- $s \equiv f(s_1, \ldots, s_n) \succ_0 g(t_1, \ldots, t_m) \equiv t$ *iff*

  (1) $\sum_{x \succsim_V y} |s|_x \geq \sum_{x \succsim_V y} |t|_x$ *for all $y \in \mathcal{V}$ and*

  (2a) $\varphi(s) > \varphi(t)$ *or*

  (2b) $\varphi(s) = \varphi(t)$, $f >_{\mathcal{F}} g$ *or*

  (2c) $\varphi(s) = \varphi(t)$, $f = g$, *and there is some k with*

  $s_1 \approx_0 t_1, \ldots, s_{k-1} \approx_0 t_{k-1}, s_k \succ_0 t_k$

- $s \succ_0 x$ *iff there exists $y \in \mathcal{V}$ with $s \succcurlyeq_{\text{st}} y \succ_V x$ or $s \succ_{\text{st}} y \succsim_V x$.*

Let $\succ$ be the corresponding KBO, that is, (1) is replaced by $|s|_x \geq |t|_x$ for all $x \in \mathcal{V}$ and $s \succ x$ iff $s \succ_{\text{st}} x$. Then $\succsim_0$ is compatible with $\succ$.

Proof Let $\succsim_0$ cover σ with respect to $\succ$. By induction on $|s| + |t|$ we show that then $s \succ_0 t$ implies $\sigma(s) \succ \sigma(t)$.

Let $t \equiv x \in \mathcal{V}$. Then there is some $y \in \mathcal{V}$ with $s \succcurlyeq_{\text{st}} y \succ_V x$ or $s \succ_{\text{st}} y \succsim_V x$. As $\succsim_0$ covers σ and the subterm ordering is stable against substitutions we get in the first

case $\sigma(s) \succcurlyeq_{\text{st}} \sigma(y) \succ \sigma(x)$ and in the second case $\sigma(s) \succ_{\text{st}} \sigma(y) \succcurlyeq \sigma(x)$, hence in both cases $\sigma(s) \succ \sigma(x)$ as $\succ_{\text{st}} \subseteq \succ$.

Let $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv g(t_1, \ldots, t_m)$. As $\succsim_0$ and $\succsim_V$ coincide on $\mathcal{V}$ and furthermore $\succsim_0$ covers σ we have for $x, y \in \mathcal{V}$ that $x \succsim_V y$ implies $\sigma(x) \succcurlyeq \sigma(y)$ and hence $\varphi(\sigma(x)) \geq \varphi(\sigma(y))$. Therefore, (1) together with $\varphi(s) \geq \varphi(t)$ implies $\varphi(\sigma(s)) \geq \varphi(\sigma(t))$. If $\varphi(\sigma(s)) > \varphi(\sigma(t))$ then $\sigma(s) \succ \sigma(t)$; otherwise $\varphi(\sigma(s)) = \varphi(\sigma(t))$. In case of $f >_{\mathcal{F}} g$ it follows that $\sigma(s) \succ \sigma(t)$. In case of $f = g$ we have by (2c) that there is some k with $s_1 \approx_0 t_1, \ldots, s_{k-1} \approx_0 t_{k-1}$, and $s_k \succ_0 t_k$. By Lemma 6.4 we get $\sigma(s_i) \equiv \sigma(t_i)$ for $i \in \{1, \ldots, k-1\}$ and by induction hypothesis we get $\sigma(s_k) \succ \sigma(t_k)$, which together imply $\sigma(s) \succ \sigma(t)$. □

As in Lemma 6.5 the orderings $\succsim_0$ and $\succcurlyeq$ coincide if $\succsim_V = \equiv$. Then condition (1) reduces to the ordinary variable condition of KBO. Nevertheless, this condition is the most complicated part in the definition of $\succsim_0$. It reflects the interaction of substitutions and weights in the KBO, which is illustrated by the following example.

Example 6.7 *Let $\varphi(f) = \varphi(a) = \varphi(b) = 1$, $\varphi(x) = 1$ for all $x \in \mathcal{V}$, and $f >_{\mathcal{F}} a >_{\mathcal{F}} b$. Consider $s \equiv f(x, a)$ and $t \equiv f(y, y)$ (as in Example 6.6). Under the constraint $x \succ_V y$ the terms s and t are incomparable by condition (1). For variable x the sum evaluates to 1 for s and to 0 for t. This rules out $t \succsim_0 s$, which is something one expects under the constraint $x \succ_V y$, as there are covered substitutions σ with $\varphi(\sigma(x)) > 2\varphi(\sigma(y)) - \varphi(a)$. For variable y, however, the sum evaluates to 1 for s and to 2 for t, which rules out the other direction. One example for this are covered substitutions σ with $\varphi(\sigma(x)) = \varphi(\sigma(y)) > \varphi(a)$.*

Considering the constraint $x \approx_V y$ terms s and t are incomparable, too. The case $s \succ_0 t$ is ruled out by condition (1), consider for example $\sigma(x) \equiv \sigma(y) \equiv f(a, a)$. The case $t \succsim_0 s$ reduces to the comparison of y and a. These are incomparable because there exist covered substitutions σ_1, σ_2 with $\sigma_1(y) \succ a$ and $a \succ \sigma_2(y)$.

Only with constraint $y \succ_V x$ the terms s and t are comparable. Then we have $t \succ_0 s$ as condition (1) is fulfilled, the weights are the same and the lexicographic comparison is determined by $y \succ_0 x$. A typical covered substitution is $\sigma = \{x \mapsto a, y \mapsto f(a, a)\}$. Because there are covered substitutions σ with $\sigma(y) \succ \sigma(y)$ and $\varphi(\sigma(y)) = 1$ for this case it is important that $\varphi(a) = 1$ (i. e., the minimal weight). For the case $\varphi(a) > 1$ the weight condition is violated, which would indicate that for such covered substitutions $\sigma(t) \not\succ \sigma(s)$ would hold.

Constructing small sets of covering orderings

For both LPO and KBO the compatible ordering $\succsim_0$ and the quasi-ordering on the variables $\succsim_V$ coincide on $\mathcal{V}$ and therefore cover the same ground substitutions. Thus, we can concentrate on $\succsim_V$ to answer question (2). The idea is to fix a set $V_0 \subseteq \mathrm{Var}(s, t)$ and to consider all total quasi-orderings $\succsim_{V_i}$ on V_0.

Lemma 6.7 *Let $V_0 \subseteq \mathrm{Var}(s, t)$, and let O denote the set of all total quasi-orderings on V_0. Then for each $\sigma \in \mathrm{GSub}(s, t)$ there exists an ordering $\succsim_V \in O$ such that $\succsim_V$ covers σ with respect to reduction ordering $\succ$.*

PROOF The substitution σ determines with $\succ$ a quasi-ordering $\succsim_V$ on $\mathcal{V}$ via $x \approx_V y$ iff $\sigma(s) \equiv \sigma(y)$ and $x \succ_V y$ iff $\sigma(x) \succ \sigma(y)$. Obviously, $\succsim_V$ covers σ. Because $\succ$ is total on ground terms and $V_0 \subseteq \mathrm{dom}(\sigma)$, the quasi-ordering $\succsim_V$ is total on V_0. But then $\succsim_V \in O$. □

This leads to a straightforward construction of the set O of Theorem 6.3.

EXAMPLE 6.8 *If* $V_0 = \{x, y\} \subseteq \mathrm{Var}(s,t)$ *and* $\succsim_1$, $\succsim_2$, *and* $\succsim_3$ *are the extended LPOs of* $\succ$ *with* $x \succ_1 y$, $y \succ_2 x$ *and* $x \approx_3 y$ *(see Lemma 6.5), then* $O = \{\succsim_1, \succsim_2, \succsim_3\}$ *has the desired properties. To show ground joinability, each of the three cases has to be checked separately.*

If for some V_0 the test is not successful, we can either extend it or choose a completely different subset. A more sophisticated way is to interleave the joinability test with the extension of V_0:

EXAMPLE 6.9 *Let* $\mathrm{Var}(s,t) = \{x, y, z\}$. *We start with three cases, namely* $x \succ_V y$, $x \approx_V y$, *and* $y \succ_V x$. *Assume that* s *and* t *can be joined in the first and the second case, but not in the third one. We then refine the third constraint into five cases:* $z \succ_V y \succ_V x$, $z \approx_V y \succ_V x$, $y \succ_V z \succ_V x$, $y \succ_V z \approx_V x$, *and* $y \succ_V x \succ_V z$. *If then* s *and* t *are joinable under each constraint, in summary we have considered only eight cases, instead of thirteen, which we need to describe all possible relationships between three variables. Furthermore, the rewrite steps performed in the third case can be shared between the five sub-cases. The effect is of course far more pronounced when more variables occur in the equation.*

Implementation aspects

A simple way to implement variable constraints $\succsim_V$ is to introduce new constants $c_1 \succ \cdots \succ c_n$, map the ordering relation $\succsim_V$ onto a corresponding substitution $\rho : V_0 \to \{c_1, \ldots, c_n\}$, and try to join the instance $\rho(s) = \rho(t)$ with the reduction ordering modified as indicated in Lemma 6.5 and Lemma 6.6.[2] To fix the set V_0, we select two variables from $\mathrm{Var}(s,t)$ by some heuristics and extend V_0 incrementally as in Example 6.9. If the size of $\mathrm{Var}(s,t)$ is larger than a given limit, we prefer to select for V_0 different small sets, say of two or three elements, instead of continuously extending V_0. The advantage is that then the number of cases depends polynomially on the number of variables of s and t. Keeping the number of different cases small is important in order to make the test feasible in practice. As only a part of the tested equations are actually ground joinable it is furthermore important to speed up the case when the test returns a negative result. There is practical evidence that in order to fail early it is beneficial to first consider the cases that map the variables to different constants.

The method can be extended by using subsumption steps. If $s \longmapsto_E t$ holds, then $s = t$ is ground joinable in E: Let $u = v$ be the equation that subsumes $s = t$. Then,

[2] The modifications slightly increase the running times of the orderings, but the optimizations of Chapter 4 are still applicable.

ground instances with $\sigma(s) \succ \sigma(t)$ can be joined with the step $\sigma(s) \longrightarrow_{\{u=v\}^{\succ}} \sigma(t)$ and ground instances with $\sigma(t) \succ \sigma(s)$ can be joined with the step $\sigma(s) \longleftarrow_{\{u=v\}^{\succ}} \sigma(t)$. For the remaining ground instances $\sigma(s) \equiv \sigma(t)$ holds, hence Theorem 3.3 applies if the $\rhd$-conditions are fulfilled. These conditions are unproblematic if the subsumption step occurs at $p \neq \lambda$ or if before the subsumption step at least one rewrite step was performed in the larger side of the case considered. Otherwise, we have to perform an explicit $\rhd$-check between the subsuming equation and the uninstantiated equation that is subject to the ground joinability test.

The use of subsumption steps speeds up the test as it can avoid a lot of case splits caused by otherwise necessary extensions of V_0. Furthermore, subsumption steps strengthen the test as they mitigate the following weakness, which is already mentioned in [MN90]: A variable x is not necessarily comparable to a term t if $x \notin \mathrm{Var}(t)$. Hence the method may show $s=t$ ground joinable, but may fail for $\sigma(s)=\sigma(t)$. For example, the equation $x_1 + x_2 + x_3 + f(x_4) = f(x_4) + x_1 + x_2 + x_3$ cannot be shown to be ground joinable in ACC' with the criterion: If $x_4 \succsim_V x_3$, then $f(x_4) \succ_0 x_3$, but if $x_3 \succ_V x_4$, then $f(x_4)$ is in $\succ_0$ not comparable with x_3 for both LPO or KBO. Equation $x_1 + x_2 + x_3 + x_4 = x_4 + x_1 + x_2 + x_3$, however, poses no problem. In contrast, the criterion based on a ground convergent subsystem of Chapter 6.2 can easily detect both equations as ground joinable in ACC'. So none of the two criteria strictly extends the other. A technique that uses case splitting but does not have this weakness is the topic of Section 6.4.

The biggest challenge for turning this redundancy criterion into a practically useful method is the implementation of efficient backward tests. These are indispensable as we frequently observe that newly activated equations are necessary to show older equations ground joinable. Our attempts to modify the critical pair heuristics such that backward tests become unnecessary have failed completely. We therefore use the following scheme to speed up the backward tests: For each equation for which the test fails we store two *witness terms*. These are the normal forms of a nonjoinable instance. We then consider the equation for a full ground joinability test only when the newly activated equation can reduce one of the witness terms. Because of this approximation we lose opportunities to show equations redundant, but it is absolutely necessary as a pre-filter to perform the backward tests in reasonable time.

6.4 Confluence trees

Among the redundancy criteria that we present in this chapter, *confluence trees* are without doubt the strongest means to detect ground joinability. Despite the undecidability of joinability and ground joinability in an ordered rewrite system (Theorem 3.4) confluence trees allow us to decide whether an ordered rewrite system is ground confluent. For this purpose the method was originally introduced for $\succ$ being an LPO [Nie93a, CNNR98], later generalized to $\succ$ being some RPOS [CNNR03] using the method of [Nie93b] to solve RPOS-constraints. However, our focus is to use confluence trees as a redundancy criterion. Furthermore, in our exposition we will try to abstract from the underlying ordering and ordering constraint solving method.

The approach is rather involved. However, it allows us to analyze the ground reducibility problem in a most precise way. Hence, we spend some space to introduce its constituents step-by-step and take the liberty to go into some details. For notational convenience we frequently use e to denote equations instead of $s = t$. By considering equations as terms with top symbol $=$ we extend the notions of $e|_p$, $e[u]_p$, etc., and even $\succ$ from terms to equations.

Standard confluence trees

The method is based on a splitting by cases, which is expressed by equations constrained by full equality and ordering constraints (see Chapter 5). Each *constrained equation* $e \mid C$ *covers* the ground instances $\sigma(e)$ with $\sigma \in \mathrm{Sol}(C) \cap \mathrm{GSub}(e)$. For example, $x + y = a \mid f(y) > x$ describes all ground instances $t + t' = a$ with $t, t' \in \mathrm{Term}(\mathcal{F}^{\mathrm{e}})$ and $f(t') \succ t$. The constrained equation $x + y = a \mid x \geq f(y)$ describes the remaining ground instances of $x + y = a$ as the reduction ordering we use is total on ground terms.

Using constrained equations we can capture the effects of rewrite steps on the level of ground instances in a precise way.

Definition 6.2 *A* constrained rewrite step *rewrites the equation* $e \mid C$ *with equation* $u = v$ *in* $R \cup E$ *into the* rewritten equation $e[\sigma(v)]_p \mid C \wedge \sigma(u) > \sigma(v)$ *and into the* complementary equation $e \mid C \wedge \sigma(v) \geq \sigma(u)$ *iff*

- $e|_p \equiv \sigma(u)$,
- $\sigma(x) \equiv c_{\min}$ *for every* $x \in \mathrm{Var}(v) - \mathrm{Var}(u)$,
- $C \wedge \sigma(u) > \sigma(v)$ *is satisfiable.*

Some comments are in order. The first condition is the standard requirement of a match from the left-hand of the rewriting equation to the redex. The second condition reflects the handling of extra variables in the rewrite system $E_{\succ}$. Constrained rewrite steps therefore lead to uniquely determined rewritten equations – even if they use equations with extra variables for which the match does not determine the bindings (see Chapter 2, p. 11). The third condition states that there must be some ground instances that are actually rewritten by $u = v$. They become strictly smaller by the step: For all substitutions $\theta \in \mathrm{Sol}(C \wedge \sigma(u) > \sigma(v))$ we have $\theta(\sigma(u)) \succ \theta(\sigma(v))$ and therefore $\theta(e) \succ \theta(e[\sigma(v)]_p)$. Enhancing the constraint C by $\sigma(v) \geq \sigma(u)$ in the complementary equation captures exactly the ground instances that this rewrite step leaves unaffected. The complementary equation therefore covers fewer ground instances than the original one.

For a given equation $s = t$ we construct a tree in the following way: We start with a node that contains the constrained equation $s = t \mid \top$. To each node for which a constrained rewrite step is possible we attach a child node with the rewritten equation and a child node with the complementary equation. Obviously, the tree describes possible rewrite steps on the ground instances of $s = t$. We call a node $u = v \mid C$ a *tautology* iff $u \equiv v$ or C is unsatisfiable. The first case means that all covered ground instances are joined. The second case means that no ground instances are left. If all

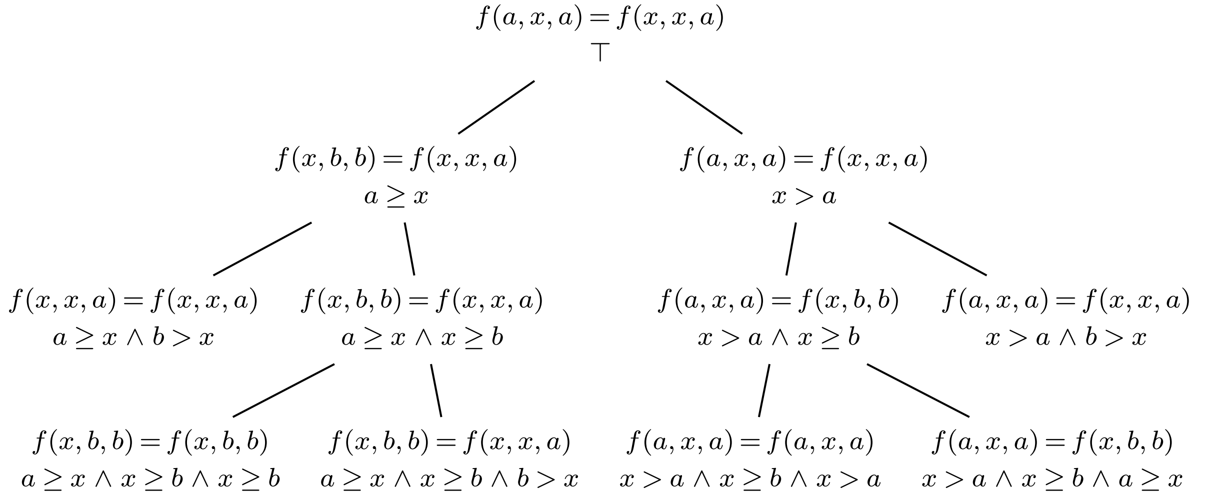


Figure 6.1: Tree for showing $f(a,x,a) \Downarrow f(x,x,a)$ in E of Example 6.10. For each constrained rewrite step the left child is the rewritten equation and the right child is the complementary equation. Constraints are depicted simplified.

leaves of the tree are tautologies we have shown for each ground instance of $s = t$ that it is joinable, hence $s \Downarrow t$.

Example 6.10 *Let $\succ$ be the LPO for precedence $f >_{\mathcal{F}} a >_{\mathcal{F}} b$. As rewrite system consider $R = \varnothing$ and $E = \{f(a,x,a) = f(x,b,b), f(x,x,a) = f(x,b,b)\}$. We want to show that $f(a,x,a) \Downarrow f(x,x,a)$ in E. We get the tree depicted in Figure 6.1.[3] All leaves that are rewritten equations have terms that are identical, whereas all leaves that are complementary equations have unsatisfiable constraints. Hence, all leaves are tautologies. Note that both equations of E occur at inner nodes as equations with some constraint. This suggests that some kind of subsumption steps may be helpful. We will come back to this later (see p. 132).*

Two differences to the criterion based on variable constraints are already evident: First, the case splits are performed *by need* depending on the elements of R and E, and not *a priori* disregarding the actual rewrite system. Second, the constraints are not limited to relations between variables; they can describe relations between arbitrary terms. This allows us to handle situations where variable constraints are too weak, such as Example 6.10 for which the method of Section 6.3 fails to show ground joinability. Especially, we can avoid an already mentioned weakness of this method, namely, that it may be unable to prove $\sigma(s) \Downarrow \sigma(t)$ even if it can show $s \Downarrow t$. The price is that we have to check the satisfiability of full ordering constraints. This problem is known to be NP-complete for $\succ$ being an LPO or a KBO (see Chapter 5).

A constraint C that contains for example the atoms $x \geq a$ and $a \geq x$ requires any solution of C to bind x to a. As the substitution σ in a constrained rewrite step

[3] For the purpose of presentation we have simplified the constraints. For $\varrho \in \{>, \geq\}$ constraint $f(a,x,a) \mathrel{\varrho} f(x,b,b)$ is equivalent to $a \geq x$, constraint $f(x,b,b) \mathrel{\varrho} f(a,x,a)$ is equivalent to $x > a$, constraint $f(x,x,a) \mathrel{\varrho} f(x,b,b)$ is equivalent to $x \geq b$, and constraint $f(x,b,b) \mathrel{\varrho} f(x,x,a)$ is equivalent to $b > x$.

is determined by matching on the equation without inspecting the constraint it is necessary to transfer such knowledge from the constraint part to the equational part. For this purpose we first introduce the following notion concerning substitutions and constraints.

Definition 6.3 *Let C be a constraint. Then an idempotent substitution σ is a* variable binding *induced by C if $\mathrm{dom}(\sigma) \subseteq \mathrm{Var}(C)$ and $\mathrm{Sol}(C) \subseteq \mathrm{Sol}(C_\sigma)$ where C_σ is the constraint $\bigwedge_{x \in \mathrm{dom}(\sigma)} x = \sigma(x)$.*

Lemma 6.8 *If σ is a variable binding induced by C then $\mathrm{Sol}(C) \subseteq \mathrm{Sol}(\sigma(C))$. Furthermore, $e \,|\, C$ and $\sigma(e) \,|\, \sigma(C)$ cover the same instances.*

Proof Let $\theta \in \mathrm{Sol}(C)$. Then $\theta \in \mathrm{Sol}(C_\sigma)$ by Definition 6.3. Hence, $\theta(x) \equiv \theta(\sigma(x))$ for all $x \in \mathrm{dom}(\sigma)$. Therefore, $\theta(t) \equiv \theta(\sigma(t))$ for all $t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$, which implies $\theta(C) \equiv \theta(\sigma(C))$ and $\theta(e) \equiv \theta(\sigma(e))$. Hence, $\theta \in \mathrm{Sol}(\sigma(C))$ and the instance $\theta(e)$ is covered by $\sigma(e) \,|\, \sigma(C)$. Conversely, let $\rho \in \mathrm{Sol}(\sigma(C))$. Then $\rho \circ \sigma \in \mathrm{Sol}(C)$. Therefore, the instance $\rho(\sigma(e))$ is covered by $e \,|\, C$. □

Many methods that test the satisfiability of constraints offer an easy way to determine for a satisfiable constraint an induced variable binding.[4] We assume that function $\mathrm{ivb}(C)$ returns such a substitution. The weakest result that we can get from $\mathrm{ivb}(C)$ is substitution id, which gives us no information as $\mathrm{id}(t) \equiv t$ for all $t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. If C consists only of equality constraints, then the most general unifier is the strongest result that $\mathrm{ivb}(C)$ can return. If C contains ordering constraints, then $\mathrm{ivb}(C)$ may be influenced by the underlying reduction ordering $\succ$. This is demonstrated by the following example.

Example 6.11 *Let C be the constraint $f(x) > g(y) \wedge g(y) \geq x \wedge x > z$. If $\succ$ is some LPO with $f >_{\mathcal{F}} g >_{\mathcal{F}} a$, then $\mathrm{ivb}(C) = \mathrm{id}$ must hold, as there are for example the solutions $\sigma_1 = \{x \mapsto g(a), y \mapsto g(g(a)), z \mapsto a\}$ and $\sigma_2 = \{x \mapsto f(a), y \mapsto f(a), z \mapsto g(a)\}$. If, however, $\succ$ is some LPO with $g >_{\mathcal{F}} f >_{\mathcal{F}} a$, then $\mathrm{ivb}(C) = \{x \mapsto g(y)\}$ is possible, as $f(x) > g(y)$ is then equivalent to $x \geq g(y)$ which together with $g(y) \geq x$ implies $x = g(y)$.*

We are now able to formulate a second kind of steps.

Definition 6.4 *An* instantiation step *rewrites the constrained equation $e \,|\, C$ into the constrained equation $\sigma(e) \,|\, \sigma(C)$ iff C is satisfiable, $\sigma = \mathrm{ivb}(C)$, and $\sigma \neq \mathrm{id}$.*

Instantiation steps are frequently necessary to show some equation ground joinable.[5] As can be seen in the next example, they are especially useful for complementary equations that are satisfiable and for which no constrained rewrite step is applicable.

[4] For the method of Section 5.2 it is easy to convert the binding context B into an idempotent substitution. Similar, for the generic method of Section 5.4 it is possible to derive such a substitution from the saturated table $\mathfrak{C}$ with the help of table $\mathfrak{D}$. However, the tests of Section 5.3 do not offer such a facility.

[5] In fact, it is rather difficult to come up with examples that do not need them, such as Example 6.10.

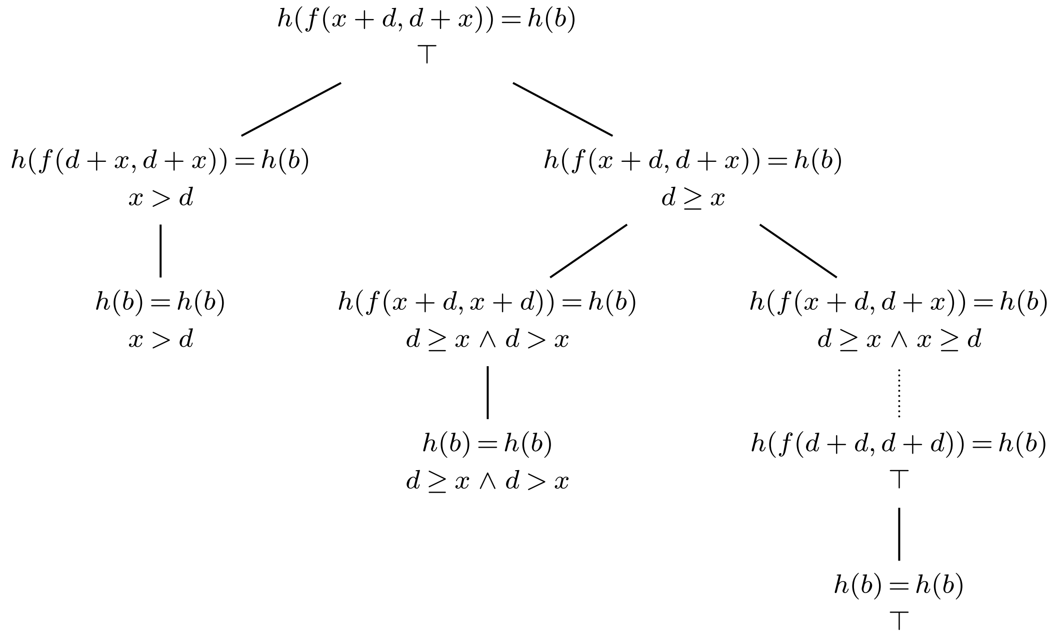


Figure 6.2: Tree for showing $h(f(x+d, d+x)) \Downarrow h(b)$ in R and E of Example 6.12. The dotted line indicates the instantiation step. Constraints are depicted simplified. Complementary equations for constrained rewrite steps with rules are omitted because they have unsatisfiable constraints.

Example 6.12 *Let $\succ$ be some LPO or KBO. Let R consist of the rule $f(x,x) \to b$ and E consist of the equation $x + y = y + x$. For checking $h(f(x+d, d+x)) \Downarrow h(b)$ we get the tree shown in Figure 6.2. Constrained rewrite steps with rules always have complementary equations with unsatisfiable constraint. This is easy to see, as $l \succ r$ implies that $\mathrm{Sol}(\sigma(r) \geq \sigma(l)) = \varnothing$. Furthermore, the constraint $\sigma(l) > \sigma(r)$ is then fulfilled by any substitution. Hence, we depict for these cases only the rewritten equations (with simplified constraints). Note that no constrained rewrite step is applicable in the case in which the instantiation step is performed.*

To enable instantiation steps sometimes a third kind of steps is necessary: the decomposition of a constraint C into an equivalent disjunction $C_1 \vee \ldots \vee C_n$. The following example illustrates such situations.

Example 6.13 *Let $\succ$ be an LPO with $g >_{\mathcal{F}} f$ and consider as constraint C the conjunction $x \geq y \wedge x \geq z \wedge f(g(y), g(z)) > g(x)$. As $g >_{\mathcal{F}} f$ the third inequation is equivalent to $g(y) \geq g(x) \vee g(z) \geq g(x)$, which further simplifies to $y \geq x \vee z \geq x$. Together with the first two inequations of C this means that constraint C is equivalent to $(x = y \wedge x \geq z) \vee (x = z \wedge x \geq y)$. Thus, two* different *instantiation steps are possible.*

It is therefore sensible to expand a node containing $e \,|\, C$ with C being equivalent to $C_1 \vee \ldots \vee C_n$ with the child nodes $e \,|\, C_i$, $i = 1, \ldots, n$. The question is, how to determine such disjunctions? A reasonable requirement is that $\mathrm{Sol}(C_i) \neq \varnothing$ for all $i = 1, \ldots, n$. Many algorithms for solving ordering constraints have some notion of

solved form [Com90, Nie93b, NR02]. For such solved forms it is usually easy to give a solution. We therefore require the disjunction to consist of solved forms.

DEFINITION 6.5 *A* decomposition step *rewrites a constrained equation* $e \mid C$ *into the set* $\{e \mid C_1, \ldots, e \mid C_n\}$ *iff* C *is satisfiable and not a solved form,* $\{C_1, \ldots, C_n\}$ *is a finite set of solved forms, and* $\mathrm{Sol}(C) = \mathrm{Sol}(C_1 \vee \ldots \vee C_n)$.

When we compare the constraints of a node and the constraints of its children, we see that a decomposition step strengthens the constraints, because $\mathrm{Sol}(C_i) \subseteq \mathrm{Sol}(C)$ for all $i = 1, \ldots, n$.

Based on the described underlying concepts we can now define confluence trees, which combine all three kinds of steps.

DEFINITION 6.6 *A* confluence tree *for* $(R, E, \succ)$ *and a constrained equation* $s = t \mid C$ *is a tree* T *where the nodes of* T *are constrained equations, the root of* T *is* $s = t \mid C$*, and the children of a node are the constrained equations obtained by either a constrained rewrite step, or an instantiation step, or a decomposition step.*

A confluence tree is closed *if all its leaves are tautologies.*

LEMMA 6.9 *Let* $\succ$ *be a simplification ordering. Then, every confluence tree is finite.*

PROOF As every node has only a finite number of children and the tree is therefore finitely branching, König's Lemma applies. Hence, we need only to show that every branch is finite. Assume that there is an infinite branch giving the infinite sequence $(e_i \mid C_i)_{i \in \mathbb{N}}$ of constrained equations. Because instantiation steps strictly decrease the number of variables and the other steps do not increase them[6], only a finite number of instantiation steps may occur. If $e_i \mid C_i$ is result of a decomposition step, C_i is a solved form, hence $e_{i+1} \mid C_{i+1}$ must arise from a different kind of step. This means that from some point on an infinite sequence of constrained rewrite steps must occur, with single decomposition steps possibly interspersed. As decomposition steps only strengthen the constraint and leave the equation unmodified, we can ignore them. Because $R \cup E$ and $\mathcal{O}(e_i)$ are finite and extra variables are bound to $c_{\min}$, there can be only a finite number of subsequent complementary equations. Equations are not applied twice to the same position as then the constraint of the rewritten equation becomes unsatisfiable. This means that there must be an infinite subsequence of rewritten equations. The terms occurring in this subsequence are built over a finite signature and with a finite number of variables. Hence, Kruskal's Theorem applies, which means that there must be some nodes $e_i \mid C_i$ and $e_j \mid C_j$ with $i < j$ such that $e_j \succcurlyeq_{\mathrm{emb}} e_i$. As $\succ$ is a simplification ordering this implies for $\theta \in \mathrm{Sol}(C_j)$ that $\theta(e_j) \succcurlyeq \theta(e_i)$, which contradicts that ground instances become strictly $\succ$-smaller along rewritten equations. □

COROLLARY 6.2 *Let the decomposition of constraints be a deterministic function. Then there exists only a finite number of confluence trees for* $(R, E, \succ)$ *and* $s = t \mid C$. □

[6] In constrained rewrite steps extra variables are bound to $c_{\min}$.

Lemma 6.10 *Let T be a closed confluence tree for $(R, E, \succ)$ and $s = t \mid C$. Then we have $\theta(s) \downarrow \theta(t)$ for all $\theta \in \mathrm{Sol}(C) \cap \mathrm{GSub}(s,t)$.*

Proof Induction on the height of T. Let N be the root node of T. If N is a leave, then it is a tautology, hence $s \equiv t$ or $\mathrm{Sol}(C) = \varnothing$, which implies that all instances are trivially joinable. If N has children $N_1, \ldots, N_n$, it suffices to show that for all $\theta \in \mathrm{Sol}(C) \cap \mathrm{GSub}(s,t)$ instance $\theta(s) = \theta(t)$ is covered by one of the children, as then the induction hypothesis applies. The children are obtained by one of the three different kinds of steps:

In case of a constrained rewrite step with $u = v \in R \cup E$ and substitution σ, child N_1 contains the rewritten equation and child N_2 the complementary equation. As $\theta \in \mathrm{Sol}(C) \cap \mathrm{GSub}(s,t)$ either $\theta \in \mathrm{Sol}(C \wedge \sigma(u) > \sigma(v))$, which implies that then the instance $\theta(s) = \theta(t)$ rewrites to an instance of N_1, or $\theta \in \mathrm{Sol}(C \wedge \sigma(v) \geq \sigma(u))$, which implies that the instance is covered by N_2.

In case of an instantiation step, child N_1 is $\sigma(s) = \sigma(t) \mid \sigma(C)$ with $\sigma = \mathrm{ivb}(C)$. By Lemma 6.8, instance $\theta(s) = \theta(t)$ is then covered by N_1.

In case of a decomposition step, instance $\theta(s) = \theta(t)$ is covered by at least one of the children: $\theta \in \mathrm{Sol}(C) = \mathrm{Sol}(C_1 \vee \ldots \vee C_n)$ implies $\theta \in \mathrm{Sol}(C_i)$ for at least one C_i. □

Therefore, we can use confluence trees as a sufficient method to test ground joinability.

Theorem 6.4 *Let T be a closed confluence tree for $(R, E, \succ)$ and $s = t \mid \top$. Then we have $s \Downarrow t$ in $R(E)$.* □

Confluence trees as redundancy criterion: Details of the proof ordering

Up to now, we have only presented confluence trees as a means to show $s \Downarrow t$ in a rewrite system $R(E)$. However, as pointed out in Theorem 3.3 and Example 3.6 (pp. 25–26), for our purposes we need a test for $s \Downarrow_{\rhd} t$. This means that to ensure that we find a smaller proof we have to take special care for the first rewrite step on the larger side of $\sigma(s) = \sigma(t)$, $\sigma \in \mathrm{GSub}(s,t)$. Confluence trees are very detailed and allow to make very fine distinctions. Therefore, we have to understand some fine points of the proof ordering (see pp. 18–19) to see what modifications are necessary.

Consider Figure 6.2, which contains the confluence tree for Example 6.12. Here, all constrained rewrite steps occur in the larger side of the equation. However, in all three cases they occur at position $p \neq \lambda$, hence the second component of the complexity of the proof step applies: Rewriting proper subterms leads to smaller proofs. Thus, the confluence tree of Figure 6.2 does not only show $h(f(x+d, d+x)) \Downarrow h(b)$ but also $h(f(x+d, d+x)) \Downarrow_{\rhd} h(b)$.

The situation is not so clear for the following variation of Example 6.12. Instead of testing $h(f(x+d, d+x)) \Downarrow_{\rhd} h(b)$ we want to show $f(x+d, d+x) \Downarrow_{\rhd} b$ in $R = \{f(x,x) \to b\}$ and $E = \{x + y = y + x\}$. We get the confluence tree depicted in Figure 6.3. Whereas the first two constrained rewrite steps occur at proper subterms, the last constrained rewrite step (after the instantiation step) occurs at top-level. We then

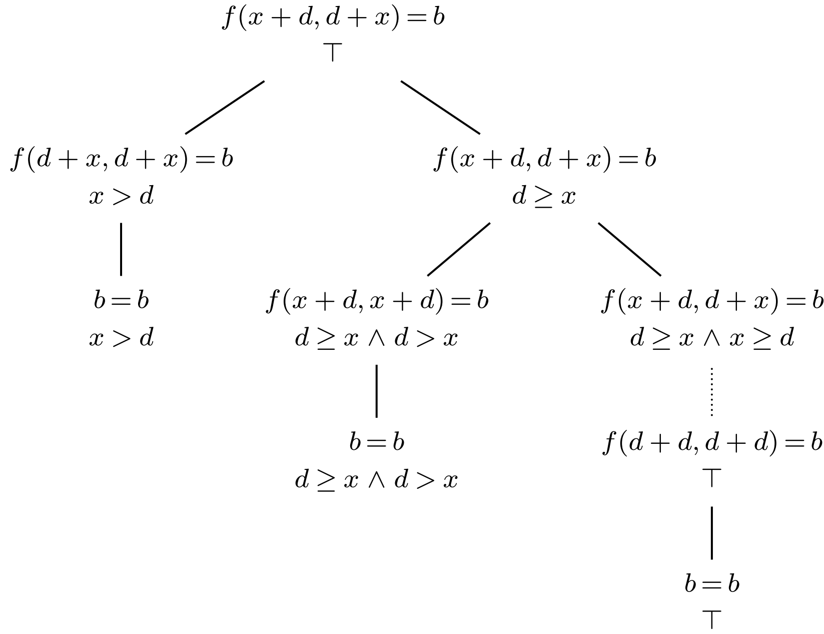


Figure 6.3: Tree for showing $f(x+d, d+x) \Downarrow_{\triangleright} b$ in $R = \{f(x,x) \rightarrow b\}$ and $E = \{x+y=y+x\}$, $\succ$ some LPO or KBO.

have to perform an explicit test with ordering $\triangleright$ which compares term pairs. Recall that we do *not* have to compare the instance $(f(d+d, d+d), b)$ with $(f(x,x), b)$, but the original, uninstantiated term pair $(f(x+d, d+x), b)$ with $(f(x,x), b)$. Here, the first component of ordering $\triangleright$ applies; as the length of $f(x,x)$ is strictly smaller than the length of $f(x+d, d+x)$, we have in fact a proof for $f(x+d, d+x) \Downarrow_{\triangleright} b$.

Note that the proof orderings of e. g. [BDH86] or [BDP89] do *not* justify this, as $f(x+d, d+x)$ and $f(x,x)$ are both incomparable by the subsumption ordering $\succ_{\text{sub}}$ and the encompassment ordering $\mathrel{\dot{\triangleright}}$. A more powerful redundancy elimination requires a more elaborate proof ordering. Examples like this have prompted us to enhance the usually used encompassment ordering (see e. g. [BDP89]) by prepending two orderings based on the term length and on the multiset of function symbols. Both ordering relations are implied by the encompassment ordering, hence the combination of the three orderings strictly *extends* the encompassment ordering. Therefore, more proof steps are comparable, which leads to more redundancy proofs. To see where the second component of $\triangleright$ is useful, replace in the example $f(x,x) \rightarrow b$ with $f(x+y, x+y) \rightarrow b$. As the multiset of function symbols $M(f(x+y, x+y)) = \{f, +, +\}$ is smaller than the multiset $M(f(x+d, d+x)) = \{f, +, +, d, d\}$, equation $f(x+d, d+x)=b$ is still redundant.

For the examples considered so far, it is always clear what the larger side of $\sigma(s)=\sigma(t)$ is, as $s \succ t$. If s and t are incomparable we have to check top-level rewrite steps on both sides of the equation. This is not only a nuisance in the implementation, but may also miss some opportunities to show redundancy. Confluence trees offer, however, a far more elegant solution. Instead of constructing a tree with root node $s=t \mid \top$ we construct two trees with root nodes $s=t \mid s>t$ and $t=s \mid t>s$. Hence, we have only to

check top-level reductions on the left-hand side of the equation. This approach reflects precisely the case distinction of Definition 3.4 and Theorem 3.3, which tells us that the remaining case $s = t \mid s = t$ is trivial.

Auxiliary steps

Confluence trees were originally introduced to decide whether an ordered rewrite system $(R, E, \succ)$ is ground confluent. If for each $s = t \in \mathrm{CP}(R, E)$ there is a closed confluence tree for $s = t \mid \top$, then $s \Downarrow t$, and $R(E)$ is ground convergent by Lemma 6.1. Conversely, if for some critical pair $s = t$ the confluence tree has a nontautology leaf, then the rewrite system $R(E)$ is not ground convergent. The three introduced kinds of steps form a minimal basis to achieve exactly this characterization. For details see [CNNR03].

However, our focus is the use of confluence trees as a redundancy criterion. During the completion process, most of the time the rewrite system is *not* ground confluent. The aim is to speed up the proof search. Hence we have to develop methods and strategies that keep the computational needs reasonably small. We therefore introduce several kinds of *auxiliary steps* to capture special situations.

A special case of constrained rewrite steps are *full rewrite steps*. These are constrained rewrite steps that have complementary equations with obviously unsatisfiable constraint. This means that the complete set of covered ground instances is rewritten. So we know a priori that subsequently we have to consider only the rewritten equation. As already mentioned in Example 6.12, rewriting with rules leads to such situations. This carries over to ordered rewrite steps with equations. For example, let E contain the commutativity axiom $x + y = y + x$ and let $\succ$ be some LPO or KBO. Then the constrained rewrite step of $g(z) + z = a \mid C$ to $z + g(z) = a \mid C$ has a complementary equation with constraint $C \wedge z \geq g(z)$, which is clearly unsatisfiable. Furthermore, some constraint solving methods allow us to extend the reduction ordering by knowledge contained in the constraint. For example let $y > a$ be part of constraint C. If we use the relation $\succ_{\mathrm{lpo}}^{C}$ of the method described in Chapter 5.2 instead of $\succ$ we can rewrite $f(y + a) = a \mid C$ to $f(a + y) = a \mid C$ by an oriented rewrite step with the commutativity axiom (cf. Lemma 5.4, p. 84). With full rewrite steps instead of constrained rewrite steps we can avoid branches in the confluence tree and costly calls to the constraint satisfiability test.

The second kind of auxiliary steps, *subsumption steps*, allows us to extend the tautology check for nodes containing $s = t \mid C$. Additionally to $s \equiv t$ we test for $s \longmapsto_E t$. If this is the case we can close the branch as we know that each ground instance can be joined in $E^{\succ}$:

LEMMA 6.11 *If $s \longmapsto_E t$ then $s \Downarrow t$ in $E^{\succ}$.*

PROOF Let $s \longmapsto_E t$ with $u = v$ at position p with substitution θ, i.e., $s|_p \equiv \theta(u)$ and $t \equiv s[\theta(v)]_p$. Then, any ground instance $\sigma \in \mathrm{GSub}(s, t)$ with $\sigma(s) \succ \sigma(t)$ can be joined by an ordered rewrite step with $\sigma(\theta(u)) \rightarrow \sigma(\theta(v))$. Conversely, if $\sigma(t) \succ \sigma(s)$, an ordered rewrite step with $\sigma(\theta(u)) \rightarrow \sigma(\theta(v))$ is possible, the equation is oriented in

the opposite direction. Finally, there may be ground instances with neither $\sigma(s) \succ \sigma(t)$ nor $\sigma(t) \succ \sigma(s)$. But then $\sigma(t) \equiv \sigma(s)$, as $\succ$ is total on ground terms. $\square$

Using subsumption steps we can therefore avoid the explicit construction of a confluence tree for $s = t \mid C$ with its rather costly constraint satisfiability checks. Furthermore, we can avoid the *search* for a confluence tree. The search process may not only construct a much larger confluence tree, it may even miss the existence of a closed confluence tree.

EXAMPLE 6.14 *Let $\succ$ be some LPO or KBO.Consider the test whether the equation $f(a+z) = f(z+a)$ is ground joinable in $R = \{f(a+y) \to a\}$ and $E = \{x+y = y+x\}$. This is easy to show with a single subsumption step. Without using subsumption, we may fail to show $f(a+z) \Downarrow f(z+a)$. Starting with root node $f(a+z) = f(z+a) \mid \top$ we first perform a full rewrite step with R. To the resulting node $a = f(z+a) \mid \top$ we then apply a constrained rewrite step with $x+y = y+x$. This leads to the rewritten equation $a = f(a+z) \mid z > a$, which can be further simplified by R to $a = a \mid z > a$, hence joined. To the complementary equation $a = f(z+a) \mid a \geq z$, however, no further step is applicable. We fail to show ground joinability, because we prefer full rewrite steps with R to constrained rewrite steps with E, a reasonable search strategy.*

In the case of extra variables, i. e., $E^{\succ} \neq E_{\succ}$ (cf. Chapter 2, p. 11), subsumption steps can even make the ground joinability test strictly more powerful, as the following example demonstrates.

EXAMPLE 6.15 *Let $\succ$ be an LPO with $f >_{\mathcal{F}} g >_{\mathcal{F}} a >_{\mathcal{F}} b$, i. e., $c_{\min} \equiv b$. Let $R = \varnothing$ and $E = \{f(x) = g(y)\}$. For the test if $f(a) \Downarrow g(a)$ the method based on standard confluence trees fails: For $f(a) = g(a) \mid \top$ the only step possible is a constrained rewrite step with $f(x) = g(y)$. This leads to the rewritten equation $g(b) = g(a) \mid f(a) > g(b)$ and a complementary equation with unsatisfiable constraint, as $f(a) \succ g(b)$. The terms $g(a)$ and $g(b)$ are irreducible by E, as $f(b) \succ g(a)$ and $f(b) \succ g(b)$, hence the constraints $f(b) > g(a)$ and $f(b) > g(b)$ are unsatisfiable. Furthermore, no instantiation step or decomposition step is possible, we end up with a nontautology leaf.*

However, it is easy to show with subsumption steps that $f(a) \Downarrow g(a)$, as we have $f(a) \longmapsto\!\!\dashv_E g(a)$.

In a similar way, we can extend the tautology test by other, stronger forms of equality. A good candidate is the use of a ground convergent subsystem $R_0(E_0)$ of R and E, the topic of Chapter 6.2. If we have for equation $s = t \mid C$ that $s =_{R_0 \cup E_0} t$, then we know that we can join any ground instance $\sigma(s) = \sigma(t)$. We can therefore avoid the search for a confluence tree with its associated costs. This method pays off, if the costs for the tests are smaller than the costs of the avoided sub-trees.

To show $s \Downarrow_{\triangleright} t$, and not only $s \Downarrow t$, all auxiliary steps have to take $\triangleright$ into account. This is only necessary if in this branch of the tree no full or constrained rewrite step has occurred in the larger side of the equation and the equation of the current node is an instance of the equation of the root node. If this is the case, for example top-level

subsumption steps have to make sure that the subsuming equation is $\rhd$-smaller than the equation of the root node.

Search strategies and other optimizations

For a given equation $s = t$ and rewrite system $(R, E, \succ)$ a finite number of confluence trees exists (see Corollary 6.2). As $R(E)$ is most often not ground confluent there might be confluence trees that show $s \Downarrow_{\rhd} t$ and others that fail to do so. Unfortunately, for a redundancy criterion it is not feasible to enumerate all confluence trees until $s \Downarrow_{\rhd} t$ can be shown.[7] Therefore, we have to develop *search strategies*.

So far we have focused on the different steps that allow us to close or to extend a branch of the tree and considered them in isolation. To derive concrete algorithms for the construction of confluence trees from this nondeterministic formulation the search strategies have to determine the order in which the different kinds of steps are tried. This means that they have to take into account the relationships between the different steps to derive an ordering on the applicable steps. Then, the first of the applicable steps is used and pursued further. If it fails (i. e., the branch cannot be closed) no other step is investigated; no backtracking occurs.

With this in mind it is clear that we should first perform tautology tests. They are rather cheap (even the more elaborate auxiliary ones) and may close the branch immediately. We should next try to use an instantiation step. Depending on the method to check the satisfiability of constraints, we either can determine $\mathrm{ivb}(C)$ right after the check of satisfiability or only for solved forms (i. e., after a decomposition step). In any case, instantiation steps should be performed as early as possible. Making knowledge that is only implicit in the constraints explicit in the terms is profitable: tautology tests and full or constrained rewrite steps do not combine the information that is present in the equation and in the constraint. Finally, full rewrite steps are preferable to constrained rewrite steps. They avoid branching and the constraints remain unchanged.

However, it is not so clear if full or constrained rewrite steps should be preferred to decomposition steps or vice versa. The former concentrate on rewriting, which is important for making progress with ground joinability proofs. The price is the creation of more complicated constraints and hence more pronounced costs for the constraint solving. The decomposition steps keep the constraints small and, when $\succ_{\mathrm{lpo}}^{C}$ is used instead of $\succ$, enable more full rewrite steps. Furthermore, they enable instantiation steps. However, the branching factor of decomposition steps is usually higher than for constrained rewrite steps, so the trees will tend to have more nodes.

We have therefore to consider two variants, one that prefers full or constrained rewrite steps and one that prefers decomposition steps. As can be seen in Figure 6.4, in both variants the confluence trees are implicit in the recursion structure, there is no

[7] This can be done by a backtracking approach: For a given node, all different possible steps are tried in succession until the current branch can be closed or no further step is applicable. We have implemented such a method to assess the detection strength of the concrete redundancy criteria that are based on confluence trees. It shows excessive runtime requirements, even with limitations on the number of expanded nodes, cf. Table 6.7 on p. 153. This coincides with our experience in strengthening the usual joinability test (see Chapter 7 on p. 164).

```
ct(e, S, U) =
  let ⟨s, t⟩ = ⟨lhs(e), rhs(e)⟩
        σ = ivb(S)
  in if   s ≡ t ∨ s ⟼⊣_E t ∨ s =_{R0∪E0} t ∨
          Sol(S ∧ u) = ∅ then
          true
     elif σ ≠ id then
          ct(σ(e), σ(S), σ(U))
     elif e ⟶_{R(E_S)} e' then
          ct(e', S, U)
     elif ∃p ∈ O(e) : ∃u ≐ v ∈ E : ∃θ :
          e|_p ≡ θ(u) ∧
          Sol(S ∧ U ∧ θ(u) > θ(v)) ≠ ∅ then
          ct(e, S, U ∧ θ(v) ≥ θ(u)) ∧
          ct(e[θ(v)]_p, S, U ∧ θ(u) > θ(v))
     elif U ≢ ⊤ then
          let 𝔖 = solvedForms(S ∧ U)
          in  ⋀_{S'∈𝔖} ct(e, S', ⊤)
     else
          false
```

(a)

```
ct(e, S, U) =
  let ⟨s, t⟩ = ⟨lhs(e), rhs(e)⟩
        σ = ivb(S)
  in if   s ≡ t ∨ s ⟼⊣_E t ∨ s =_{R0∪E0} t ∨
          Sol(S ∧ u) = ∅ then
          true
     elif σ ≠ id then
          ct(σ(e), σ(S), σ(U))
     elif U ≢ ⊤ then
          let 𝔖 = solvedForms(S ∧ U)
          in  ⋀_{S'∈𝔖} ct(e, S', ⊤)
     elif e ⟶_{R(E_S)} e' then
          ct(e', S, U)
     elif ∃p ∈ O(e) : ∃u ≐ v ∈ E : ∃θ :
          e|_p ≡ θ(u) ∧
          Sol(S ∧ U ∧ θ(u) > θ(v)) ≠ ∅ then
          ct(e, S, U ∧ θ(v) ≥ θ(u)) ∧
          ct(e[θ(v)]_p, S, U ∧ θ(u) > θ(v))
     else
          false
```

(b)

Figure 6.4: Algorithms for the computation of confluence trees: (a) Variant preferring full or constrained rewrite steps. (b) Variant preferring decomposition steps.

need to construct them explicitly. The search state is described by three parameters: e represents the current equation, S and U the current constraint, where we establish the invariant that S is a solved form. This simplifies some tests (e. g., $\mathrm{ivb}(S)$ is easy to determine) and allows us to strengthen the rewrite relation by using ordered steps with respect to $\succ^S_{\mathrm{lpo}}$: We write $R(E_S)$ for $R \cup E^{\succ^S_{\mathrm{lpo}}}$. Both variants are sufficient tests for ground joinability: If $\mathrm{ct}(s = t, \top, \top) = \mathsf{true}$ then $s \Downarrow t$ in $R(E)$. To simplify the presentation we have omitted the tests that are additionally needed for $s \Downarrow_{\triangleright} t$.

As we use confluence trees as redundancy criterion during completion, most of the tests will fail, the probability that an equation is ground joinable is rather small. Therefore, we adapt a *fail early* strategy for all variants. The aim is to keep confluence trees that cannot be closed rather small. This means that in a constrained rewrite step we investigate the complementary equation before the rewritten equation (cf. Figure 6.4). The intention is that we then detect irreducible instances earlier. In a decomposition step, we first consider the solved forms with few equality and many inequality constraints. These cover instances that are harder to simplify, as less information can be transferred from the constraint part to the equational part and less equations are applicable.

We have developed several other optimizations. The first group transfers knowledge

along the branches of the tree. It is only sensible to test an equation for joinability if the equation has been modified (i. e., is a rewritten equation or the result of an instantiation step). Similar, there is no need to check the satisfiability of the constraint of a node that results from a full rewrite step, a decomposition step, or an instantiation step, or that contains a rewritten equation. For a complementary equation, there is no need to consider positions for constrained rewrite steps for which all equations have already been tried. This is especially valuable, for it avoids many calls to the constraint solver for which we know the (negative) result beforehand. In analogy to normal form strategies, after a full or constrained rewrite step we can restrict the set of positions tested for further rewrite attempts. If we use an leftmost-outermost strategy, positions that occur left to the last full or constrained rewrite position can be neglected except the path from the root to the position. We can skip for example positions in the left-hand side of the equation if the last full or constrained rewrite step occurred in the right-hand side. Some of this information can be propagated even over several levels of the tree.[8]

A second group of optimizations concerns the constraint solver. Let $\succ$ be some LPO. Originally, confluence trees were introduced using the method of [Nie93b] to solve constraints. By using the method of [NR02] (see Chapter 5.2) we first get a faster test for satisfiability. Second, the number of solved forms is smaller with this test, which is important for decomposition steps. However, we get the solved forms of a constraint by collecting the satisfiable leaves of the search tree. Hence, some solved forms may be duplicated or covered by other solved forms. Let C_i and C_j be two solved forms. If $\mathrm{Sol}(C_i) \subseteq \mathrm{Sol}(C_j)$, we can drop C_i. Furthermore, we can merge two different solved forms in some cases. In particular, the following situation occurs quite often. Let $C_i \equiv s = t \wedge C'$ and $C_j \equiv s > t \wedge C''$. If $\mathrm{Sol}(s = t \wedge C') \subseteq \mathrm{Sol}(C'')$ and $\mathrm{Sol}(s = t \wedge C'') \subseteq \mathrm{Sol}(C')$ then C_i and C_j can be replaced by $s \geq t \wedge C''$. Note that the test $\mathrm{Sol}(C_1) \subseteq \mathrm{Sol}(C_2)$ is used internally in the method of [NR02] (see Chapter 5.2), so both optimizations are easy to implement.

Nevertheless, the costs for computing a confluence tree may become prohibitive. Unfortunately, it is not easy to estimate the costs in advance, such as is possible to some degree for the method of Chapter 6.3. We therefore limit the number of nodes that we investigate and give up if this number is exceeded. This seems to be superior to limiting the depth of the tree or giving up when a decomposition step leads to too much branching, two approaches that we have used in [AL01].

To speed up backwards tests we can use a technique that is similar to the one used for the method based on variable constraints (cf. p. 124). For each equation for which the test fails we store a nontautology leaf. We then consider the equation for a full ground joinability test only if the newly activated equation can be used to extend the leaf by a full or constrained rewrite step.

[8] Unfortunately, the transfer of knowledge complicates the algorithms, so we have omitted these optimizations in Figure 6.4 for clarity reasons.

6.5 Ground reducibility

When we compare the redundancy criteria of the previous sections we see a clear distinction: The criterion based on a ground convergent subsystem is theory-specific, the two criteria showing ground joinability by case splitting are generic. For the first one, forward tests suffice, the other two require backwards tests. The first one has polynomial costs (for the theories that we considered), the other two make case distinctions where the number of cases grows exponentially with the size of the input. The first criterion is therefore rather cheap, whereas it is difficult to take profit from the latter ones – although they are more powerful. Nevertheless, experiments have shown that both generic tests take effect especially when AC-operators are present. It seems that then the search state contains many redundant equations for which the usual interreduction mechanisms are too weak and more powerful redundancy elimination is required.

The aim of this chapter is to develop a criterion that shows in the AC-case the greater power of the generic tests while retaining the desirable properties of the first one, namely being cheap and purely forward. To do that we need a better understanding of the disadvantages of the generic tests.

First, both generic tests perform too many case distinctions. We restrict therefore in practice the method based on variable constraints to equations with at most 5 different variables. Confluence trees introduce the case distinctions by need. However, they tend to make very fine distinctions, even for small examples we can observe trees with hundreds or thousands of nodes. So their costs do not originate in the use of ordering constraints, but in the sheer number of cases. Second, in both generic tests much work is duplicated between independent tests of different equations: Consider two equations that differ only in two subterms s and t that are exchanged by commutativity. As both tests have to consider the cases when s is greater than t and vice versa, it is likely that both tests exchange in some sub-cases s and t and so perform (nearly) identical work twice. Third, the tests check for complete joinability proofs only. They fail if an equation that is necessary for some ground instance is not yet available. However, the completion process does not enumerate the equations in the order that is most suitable for the tests and our attempts to develop corresponding strategies failed. This is the reason why we have to perform backwards tests and why we can take only part of the possible benefit. Finally, as demonstrated in Example 6.14, the search for a ground joinability proof may fail to find a proof even if one exists. The rewrite system is not ground confluent most of the time and the costs prevent us to widen the search by using more backtracking. To sum up, both generic tests perform too much work at once for testing redundancy during completion. This is not surprising, they were originally designed to check properties of static rewrite systems.

The main idea to reduce the work is to use *ground reducibility*, which is more localized than ground joinability. Hence, we no longer search for complete joinability proofs for each ground instance, but focus on the first step instead. To complete the proofs, we add the necessary overlaps, which in turn make the original equation redundant. This is only a win if the overlaps do not outnumber the critical pairs. In some sense, we leave the framework of redundancy criteria: Instead of detecting equations redundant, we *make* equations redundant.

DEFINITION 6.7 *An equation $s = t$ is* ground reducible *with respect to $R(E)$ if for each $\sigma \in \mathrm{GSub}(s,t)$ at least one of $\sigma(s)$ or $\sigma(t)$ is reducible by $R(E)$.*

For ordered rewriting ground reducibility is undecidable in general [CNNR03], therefore we have to use sufficient tests. We can distinguish three kinds of rewrite steps, which differ in the position at which they occur. If the position is a nonvariable position in the uninstantiated term, the step happens either at the *skeleton*, that is, all function symbols of the left-hand side of the rewriting equation match function symbols in the term, or it happens at the *fringe*, such that a part of the function symbols matches functions symbols in the substitution part. Furthermore, the step may happen completely in the *substitution part*, which means that the substitution itself is reducible. It is not necessary for theorem proving to consider such substitutions. Regarding the two other kinds of steps, if $\sigma(s) = \sigma(t)$ is reduced with $l = r$ to $s_1 = t_1$, then there is some overlap $u = v$ from $l = r$ into $s = t$ such that $s_1 = t_1$ is an instance of $u = v$.

To get smaller proofs rewrite steps at top-level have to be performed with $\rhd$-smaller equations. This explains the additional condition in the following theorem, which provides the foundation of the criterion.

THEOREM 6.5 *Let $s = t$ be ground reducible with respect to $R(E)$ such that all top-level reductions of maximal sides of ground instances are performed with equations in $R \cup E$ that are strictly $\rhd$-smaller than $s = t$. Let $S = \mathrm{OL}(s = t, R \cup E)$. Then $s = t \succ_{\mathcal{P}} R(E \cup S)$.*

PROOF Let $\sigma \in \mathrm{GSub}(s,t)$ be $R(E)$-irreducible. The case $\sigma(s) \equiv \sigma(t)$ is trivial. Consider $\sigma(s) \succ \sigma(t)$. Let P_0 be the proof $\sigma(s) \longrightarrow_{\{s=t\}^{\succ}} \sigma(t)$. As $s = t$ is ground reducible, $\sigma(s)$ or $\sigma(t)$ are reducible by $R(E)$. First, assume $\sigma(s)$ reduces to s_1 at position p with $l = r$ in $R(E)$. Let P_1 be the proof $\sigma(s) \longrightarrow_{R(E)} s_1 \longleftrightarrow_S \sigma(t)$. The first step is smaller than the proof step in P_0: If it does not occur on top-level, its complexity is smaller in the second component, otherwise in the third component, as then $(s,t) \rhd (l,r)$ by assumption. The second step of P_1 is smaller than the step in P_0 by the first component of the complexity as both $\sigma(t)$ and s_1 are smaller than $\sigma(s)$. Therefore, $P_0 \succ_{\mathcal{P}} P_1$. Second, consider the case that $\sigma(t)$ reduces to t_1 by $R(E)$. Then, $\sigma(s) \succ \sigma(t) \succ t_1$. Let P_2 be the proof $\sigma(s) \longrightarrow_S t_1 \longleftarrow_{R(E)} \sigma(t)$. The first step of P_2 is smaller than the proof step in P_0 by the last component of its complexity, as $\sigma(t) \succ t_1$. The second step is smaller by the first component of the complexity, which implies $P_0 \succ_{\mathcal{P}} P_2$.

The case $\sigma(t) \succ \sigma(s)$ is similar to the case $\sigma(s) \succ \sigma(t)$. □

As is clear from the proof, we can restrict the set S to contain the necessary overlaps only. However, depending on the available ground joinability test, it might be difficult to extract from all overlaps the necessary ones. Furthermore, depending on the prover architecture, it might be difficult to transfer such knowledge from the side of the ground reducibility test to the side of the generation of overlaps.

The patterns of proofs P_1 and P_2 resemble their counterparts used for justifying interreduction. There we can also distinguish between simplification of larger and smaller sides. The difference is, of course, that in contrast to the usual interreduction the criterion works on the ground level. Considering the smaller sides enables more redundancy

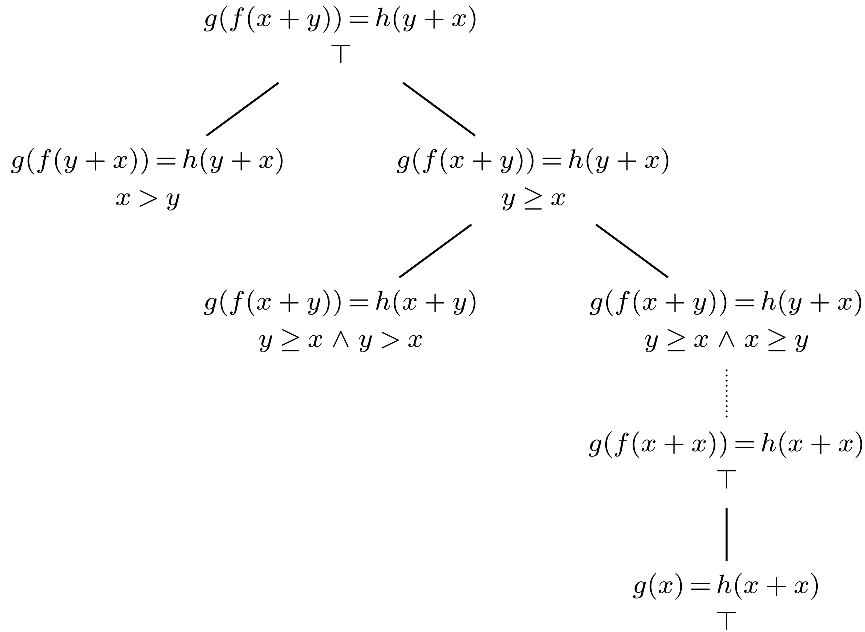


Figure 6.5: Pruned confluence tree to show in $R=\{f(x+x)\to x, f(x+a)\to a\}$ and $E=\{x+y=y+x\}$ ground reducibility of $g(f(x+y))=h(y+x)$, see Example 6.16. Constraints are depicted simplified.

proofs, at the cost of overlapping into smaller sides. This is unproblematic as long as the number of overlaps is rather small, which holds true for the special case we consider in the next section. Before that, we want to show how to handle the generic case.

With a small modification we can use confluence trees as a sufficient method to detect ground reducibility. The idea is to stop the expansion for nodes that cover rewritten instances. This means that after a constrained rewrite step only the complementary equation is considered further. After a full rewrite step, which is possible e. g. after an instantiation step, we can stop the expansion. To distinguish the resulting trees from ordinary confluence trees we call them *pruned confluence trees*. It is clear that an equation is ground reducible if we can construct a pruned confluence tree that has only leaves that cover rewritten instances. The additional condition for top-level reductions of Theorem 6.5 is handled by the usual treatment of $\rhd$ in confluence trees. Unfortunately, the necessary overlaps cannot be determined solely by analyzing the nontautology leaves as instantiation steps may have modified the terms. Instead, for each full or constrained rewrite step we have to record the used equation and the position of the redex.

EXAMPLE 6.16 *Let $\succ$ be the LPO for $+>_{\mathcal{F}} f>_{\mathcal{F}} g>_{\mathcal{F}} h>_{\mathcal{F}} a$. Consider the rewrite system $R=\{f(x+x)\to x, f(x+a)\to a\}$ and $E=\{x+y=y+x\}$. Then equation $g(f(x+y))=h(y+x)$ is ground reducible in $R(E)$. The corresponding pruned confluence tree is depicted in Figure 6.5. Note that it is necessary to consider rewrite steps in $h(y+x)$, the smaller side of the equation, to detect ground reducibility. The set S of overlaps is given by $S=\{g(x)=h(x+x),\ g(a)=h(a+x),\ g(f(y+x))=h(y+x),$*

$g(f(x+y)) = h(x+y)\}$. The second of these equations is not necessary to establish redundancy. Note that in $R(E)$ equation $f(x+y) = h(y+x)$ is not *redundant by Theorem 6.5, because any ground reducibility proof violates the $\rhd$-condition.*

It is clear that the construction of such pruned confluence trees is much cheaper than the construction of full confluence trees. Many of their disadvantages are avoided by the more localized nature of the test; the search for full ground joinability proofs is avoided. However, we still have to perform backward tests. Furthermore, the implementation effort for pruned confluence trees is nearly the same as for full confluence trees. We can do much better for the special situation we consider next.

6.5.1 The AC-case

An analysis of the successful ground joinability tests with the method based on variable constraints (see Chapter 6.3) reveals that the constructed proofs often start with an ordered reduction step with C or C'. Both equations are rather small, hence likely to be applicable, and we can often avoid the generation of other unorientable equations by a proper selection of the ordering [HJL99]. Furthermore, as already mentioned, the search state seems to contain many redundant equations especially in the AC-case.

Hence, we concentrate on ground reducibility with respect to ACC'. For $\succ$ being some LPO or KBO the simplification of ground terms with ACC' resembles bubble-sort: First, rewrite steps with A bracket the AC-subterms to the right. Then, rewrite steps with CC' exchange adjacent subterms such that smaller terms come to the left. Possible skeleton steps differ for the cases A and CC'. As A is a linear rule, skeleton steps with A imply that the equation is already A-reducible, which is easy to decide. Recall that both occurrences of operator $+$ in the left-hand side of $(x+y)+z \rightarrow x+(y+z)$ match operators in the skeleton. Hence, operators in the substitution part cannot affect the applicability of the rule at a skeleton position. For rewrite steps with CC', however, the situation is completely different. Here, the ordering relations between the instantiated subterms determine reducibility. Such ordering relations can be captured conveniently by ordering constraints. For example, we can consider C and C' as *constrained rules*[9]:

$$\begin{array}{rl} x+y \rightarrow y+x & \mid x > y \\ x+(y+z) \rightarrow y+(x+z) & \mid x > y \end{array}$$

However, for our purposes it is not necessary to develop the whole machinery of constrained rewriting. The main point is that for the skeleton case CC'-ground reducibility can be reduced to the satisfiability of ordering constraints. This becomes clear in the following example.

Example 6.17 *Let $\succ$ be some LPO or KBO. Consider equation $x+y = g(y)+(z+x)$, $+ \in \mathcal{F}$. An instance σ is C-reducible if $\sigma(x) \succ \sigma(y)$, $\sigma(g(y)) \succ \sigma(z+x)$, or $\sigma(z) \succ \sigma(x)$. It is C'-reducible if $\sigma(g(y)) \succ \sigma(z)$. Therefore, a CC'-irreducible ground instance must satisfy the constraint $y \geq x \wedge z+x \geq g(y) \wedge x \geq z \wedge z \geq g(y)$. As this constraint is unsatisfiable, we know that the equation is ground reducible.*

[9] The constraints are simplified as we assume $\succ$ to be some LPO or KBO.

Note that in the example we have two atomic constraints for the top-level operator of the right-hand side (the second and the last). Both equations, C and C', may be applicable at this position. However, each instance that can be reduced by C can also be reduced by C'. This is reflected by the corresponding constraints: $z + x \geq g(y)$ is weaker than $z \geq g(y)$ in that $\mathrm{Sol}(z \geq g(y)) \subseteq \mathrm{Sol}(z + x \geq g(y))$. Hence, we can optimize the constraint and omit $z + x \geq g(y)$. As this is a general observation, we can omit the atomic constraint corresponding to C in all cases where both C and C' are applicable. The resulting constraint contains then exactly one atomic constraint for each position where an AC-operator occurs.

The following function Γ constructs an ordering constraint for a term t such that the satisfying substitutions describe ground instances that are CC'-irreducible at skeleton positions. It is defined as follows:

$$\begin{array}{lll}
\Gamma(x) & = \top & \\
\Gamma(f(t_1,\ldots,t_n)) & = \Gamma(t_1) \wedge \ldots \wedge \Gamma(t_n) & \text{if } f \notin \mathcal{F} \\
\Gamma(t_1 + (t_1 + t_2)) & = \Gamma(t_1 + t_2) & \text{if } + \in \mathcal{F} \\
\Gamma(t_1 + (t_2 + t_3)) & = t_2 \geq t_1 \wedge \Gamma(t_1) \wedge \Gamma(t_2 + t_3) & \text{if } + \in \mathcal{F} \quad \text{and } t_1 \not\equiv t_2 \\
\Gamma(t_1 + t_1) & = \Gamma(t_1) & \text{if } + \in \mathcal{F} \quad \text{and } \mathrm{top}(t_1) \neq + \\
\Gamma(t_1 + t_2) & = t_2 \geq t_1 \wedge \Gamma(t_1) \wedge \Gamma(t_2) & \text{if } + \in \mathcal{F} \quad , t_1 \not\equiv t_2, \text{ and} \\
 & & \quad \mathrm{top}(t_2) \neq +
\end{array}$$

We write $\Gamma(s,t)$ as shorthand notation for $\Gamma(s) \wedge \Gamma(t)$.

Lemma 6.12 *If $\Gamma(t)$ is unsatisfiable, then t is CC'-ground reducible.*

Proof Let $\sigma \in \mathrm{GSub}(t)$ and $\Gamma(t) = t_1 \geq t_1' \wedge \ldots \wedge t_n \geq t_n'$. As $\Gamma(t)$ is unsatisfiable, substitution σ satisfies the complement of $\Gamma(t)$, which is equivalent to the constraint $t_1' > t_1 \vee \ldots \vee t_n' > t_n$. Therefore, there is at least one i such that $\sigma(t_i') \succ \sigma(t_i)$, which implies that C or C' are applicable at the corresponding position in $\sigma(t)$. □

Note that function Γ does not capture AC-ground reducibility in a complete way, it considers only skeleton steps. In addition, there are two kinds of fringe steps. Situations such as $x + s$ and $\sigma(x) \equiv s_1 + s_2$ lead to reduction steps with A. Reduction steps with C' are possible for subterms of the form $s + x$ if $\sigma(x) \equiv s_1 + s_2$ and $\sigma(s) \succ s_1$. Both situations can also be described by ordering constraints. But these need (local) quantification, which means that we leave (for LPO) the decidable fragment [CT97]. Finally, there are examples, such that each ground instance that is not skeleton reducible has a reducible substitution. Here, irreducibility constraints are appropriate. Deciding AC-ground reducibility with this approach requires the extension of decision procedures for the satisfiability of ordering constraints to cope with the additionally needed constraints. This is a topic of future research.

Our main focus, however, is on a practical redundancy criterion, not on a decision procedure for detecting AC-ground reducibility. To guarantee polynomial behavior, we use the test for unsatisfiability of constraints of Section 5.4, which is only sufficient. Hence, we consider only skeleton reductions. In practice we can expect the input of the redundancy test in A-normal form. We can therefore concentrate on the CC'-skeleton

case. When we combine Theorem 6.5 and Lemma 6.12 and consider that rewriting with C and C' always fulfills the $\rhd$-condition because both equations are “distinguished” (see p. 116) we get:

THEOREM 6.6 *Let $s=t$ be an equation different from C and C' such that $\Gamma(s,t)$ is unsatisfiable. Let $S = \mathrm{OL}(s=t, CC')$. Then $s=t \succ_{\mathcal{P}} CC' \cup S$.* □

The set S is very easy to compute as only subterms are exchanged and even unification is unnecessary. The size of this set is linear in the number of AC-operators in $s=t$. This means a huge improvement in practice, as only say 5 or 10 equations are constructed. For challenging proof tasks the typical size of the set of critical pairs with an equation is in the order of several thousand equations. We can restrict the set S to the necessary overlaps by using the facility of the generic unsatisfiability test for ordering constraints to determine the unsatisfiable kernel of atomic constraints (see p. 103). This allows us to reduce the number of overlaps to approximately 50 %. However, this further improvement is rather small compared to the typically thousandfold main reduction.

The generation of set S is the price to pay to avoid the many case distinctions of the generic redundancy tests and the associated duplicated work between independent tests of similar equations. By inserting S into $\mathfrak{P}$ (nearly) identical work is identified by identical equations, the usual simplification mechanisms avoid the duplication of work. Therefore, the criterion fits the usual completion approach much better. Like the redundancy criteria of Section 6.2 this criterion can be applied in a purely forward way.

Another advantage to the generic ground reducibility test that is based on pruned confluence trees is the fact that only one constraint has to be checked for satisfiability. By using the polynomial test for unsatisfiability of constraints (see Section 5.4) we have therefore achieved a very cheap criterion that avoids all disadvantages of the generic redundancy tests. Of course, it is weaker than these tests, but in practice it covers most of the redundant equations that the other tests detect, especially if we use its extension which we consider next.

6.5.2 A refined criterion for the AC-case

Our motivation to refine the criterion results from the following example.

EXAMPLE 6.18 *Let $\succ$ be some LPO or KBO. Consider equation $x+(y+x)=a$. It cannot be shown redundant by the method of the previous section, because there are CC'-irreducible ground instances. These are characterized by $\sigma(x) \equiv \sigma(y)$, that is, they unify nonidentical subterms.[10] Nevertheless, the equation seems to be redundant with respect to its CC'-overlaps, which are $x+(x+y)=a$ and $y+(x+x)=a$: both overlaps cover these instances, and both overlaps have irreducible ground instances that do not unify subterms.*

[10] It is tempting to replace the equation by $x+(x+x)=a$ in this situation. But this would endanger completeness as it would interact badly with interreductions. Formally, this is reflected by a violation of the proof ordering for such kind of steps.

It is easy to exclude such unifying solutions by a modified constraint construction. Function $\hat{\Gamma}$ is identical to Γ, except that it uses $>$-constraints instead of $\geq$-constraints. However, the proof ordering does not justify this approach. In Example 6.18 none of the two overlaps is $\rhd$-smaller than the original equation. This is not without reason, as the following example shows.

EXAMPLE 6.19 *Let $\succ$ be the LPO for $h >_{\mathcal{F}} g >_{\mathcal{F}} f$ and consider the equation $f(h(x) + g(h(y)), h(y) + g(h(x))) = x + y$. For this equation the CC'-irreducible ground instances unify x and y, as for $\succ$ the constraint $g(h(y)) \geq h(x) \wedge g(h(x)) \geq h(y) \wedge y \geq x$ is equivalent to $h(y) \geq h(x) \wedge h(x) \geq h(y) \wedge y \geq x$, which implies $x = y$. However, there is no CC'-overlap that covers these instances and that has irreducible instances that do not unify x and y.*

To retain the original idea to use $\hat{\Gamma}$ we need therefore an additional guard in the following predicate. Its definition is rather ad-hoc and technical, but the use of $\hat{\Gamma}$ is rewarding in practice. Let $S \quad (s = t) = \{s' = t' \mid s = \quad s' \text{ and } t = \quad t'\}$.

DEFINITION 6.8 *Let $s, t \in \mathrm{Term}(\mathcal{F}, \mathcal{V})$. Predicate U is given by $U(s,t) = 1$ if $s = t$ not in CC', $\Gamma(s,t)$ is satisfiable, $\hat{\Gamma}(s,t)$ is unsatisfiable, and for all $\sigma \in \mathrm{GSub}(s,t)$ that satisfy $\Gamma(s,t)$ there is some $s' = t'$ in $S \quad (s = t)$ such that $\hat{\Gamma}(s', t')$ is satisfiable and $\sigma(s) = \sigma(t)$ is an instance of $s' = t'$. Otherwise, $U(s,t) = 0$.*

The idea of that definition is that proof steps using an equation $s = t$ with $U(s,t) = 1$ can be replaced by proof steps using an equation $s' = t'$ with $U(s', t') = 0$. Equations with $U(s,t) = 1$ are therefore redundant. In other words, for every CC'-irreducible ground instance of $s = t$ there is some $s' = t'$ that covers the instance and is not declared redundant by the criterion.

LEMMA 6.13 *Let $\sigma(s) = \sigma(t)$ be a CC'-irreducible ground instance of $s = t$. Then there is some $s' = t'$ in $S \quad (s = t)$ such that $\sigma(s) = \sigma(t)$ is an instance of $s' = t'$ and $U(s', t') = 0$.*

PROOF The case $U(s,t) = 0$ is trivial. Consider $U(s,t) = 1$. Then there exists by definition of U some $s' = t'$ in $S \quad (s = t)$ such that $\sigma(s) = \sigma(t)$ is an instance of $s' = t'$ and $\hat{\Gamma}(s', t')$ is satisfiable. Therefore, $U(s', t') = 0$. □

Consider Example 6.18 again. Predicate U evaluates to 1 for the original equation and to 0 for both overlaps, because their $\hat{\Gamma}$-constraints are satisfiable. For the equation of Example 6.19, however, U evaluates to 0.

With predicate U we modify the proof ordering by modifying ordering $\rhd$ on term pairs. We extend function Ψ to Ψ' with $\Psi'(s,t) = (|s|, M(s), U(s,t), s, n)$ and $\rhd_{\Psi'}$ is the lexicographic combination $\rhd_{\Psi'} = (>_{\mathbb{N}}, \supset, >_{\mathbb{N}}, \unrhd, >_{\mathbb{N}})$. As $\rhd'$ is still well-founded, the resulting ordering $\succ_{\mathcal{P}'}$ is indeed a proof ordering.

LEMMA 6.14 *Let $s = t$ be an equation with $U(s,t) = 1$. Then for each proof step P_σ with $\sigma(s) \longleftrightarrow_{\{s = t\}} \sigma(t)$ there is some proof P in $\mathsf{CC}' \cup S \quad (s = t)$ such that $P_\sigma \succ_{\mathcal{P}'} P$ and P uses only equations $u = v$ with $U(u,v) = 0$.*

Proof The case $\sigma(s) \equiv \sigma(t)$ is trivial. Consider $\sigma(s) \succ \sigma(t)$, which means that P_σ is $\sigma(s) \longrightarrow_{\{s=t\}\succ} \sigma(t)$. Let $S = S \quad (s=t)$. Let P be the proof of $\sigma(s) \stackrel{!}{\longrightarrow}\ _{,\succ} s_1 \longleftrightarrow_S t_1 \stackrel{!}{\longleftarrow}\ _{,\succ} \sigma(t)$. Because $s_1 = t_1$ is CC'-irreducible, there is some $s' = t'$ in S such that $U(s', t') = 0$ and $s_1 = t_1$ is a ground instance of $s' = t'$. The other steps use C and C', for which U evaluates to 0. To show $P_\sigma \succ_{\mathcal{P}'} P$ we have to consider two cases for the first step. If $\sigma(s) \succ s_1$ then $\sigma(s)$ gets reduced. The complexity of the first step is therefore smaller than the complexity of the step in P_σ, either by the second component (if the rewrite step is not top-level), or by the third component. If $\sigma(s) \equiv s_1$ the first step is performed top-level with $s' = t'$, so we have to consider the third component of the complexity. We have $(s, t) \rhd' (s', t')$ as $s \equiv s'$ and $U(s, t) > U(s', t')$. The remaining steps are smaller by the first component, as $\sigma(s)$ is the maximal term in P.

The case $\sigma(t) \succ \sigma(s)$ is similar. □

This means that we can refine the criterion based on CC'-ground reducibility to additionally detect equations $s = t$ with $U(s, t) = 1$.

Theorem 6.7 *Let $s = t$ be an equation with $U(s, t) = 1$. Then $s = t \succ_{\mathcal{P}'} CC' \cup S \quad (s = t)$.* □

Note that the complete set $S \quad (s = t)$ is usually not necessary, it suffices to guarantee the existence of all needed smaller proofs. Similarly, the formulation of predicate $U(s, t)$ follows theoretical considerations. For practical purposes we use the following approach. From the unsatisfiability test we can determine an induced variable binding $\sigma = \text{ivb}(\Gamma(s, t))$ (see p. 127). If $\sigma \neq \text{id}$ and $\hat{\Gamma}(s, t)$ is unsatisfiable we then search for an equation $s' = t'$ in $S \quad (s = t)$ that subsumes $\sigma(s) = \sigma(t)$ and has a satisfiable constraint $\hat{\Gamma}(s', t')$. Hence, we search for *one* equation to cover *all* irreducible ground instances. In the definition of U *several* covering equations are possible, because the covering equation may depend on the ground instance.

For the search, it is usually not necessary to enumerate the whole set. Instead, we construct in a breadth-first manner the CC'-overlaps of $s = t$, then the CC'-overlaps of the CC'-overlaps and so forth. The unsatisfiable test offers the facility to determine the unsatisfiable kernel of the constraints. We can therefore restrict the overlaps we compute to the ones for which at least one of the atomic constraints of the unsatisfiable kernel is changed by the overlapping.

In case of $\sigma = \text{id}$, we simply give up. This occurs rarely in practice because we frequently encounter equations with "twisted" variables, such as the equation of Example 6.18. For such examples the unsatisfiability test usually determines an induced variable binding σ with $\sigma \neq \text{id}$.

Another approach is possible if additionally to ACC' also the idempotence axiom I and its extension I' are present. This avoids to test the conditions of $U(s, t)$. Instead, we use $\hat{\Gamma}$ to determine redundancy extending the approach to $CC'II'$-ground reducibility. Therefore, we have to compute additional overlaps with the idempotence axiom and its extension. These overlaps lead to shorter equations, which might be beneficial for the proof search.

In some sense, the definition of U is somewhat arbitrary and mostly motivated by suggestive examples such as Example 6.18 which encourage the strengthening of Γ to

$\hat{\Gamma}$. For completeness reasons, it is sufficient to cover for each CC'-equivalence class the CC'-irreducible ground instances in some way. A better analysis of the structure of the CC'-equivalence may help to find better solutions. This is a topic of future research.

6.6 Experimental evaluation

To evaluate the redundancy criteria of the previous sections, we implemented them into the theorem prover WALDMEISTER [LH02]. We then performed test runs on machines equipped with Pentium III 1 GHz-processors and 4 GByte RAM. As a basis for our experiments we chose from the 722 unsatisfiable unit equality problems of the TPTP Version 2.7.0 [SS98] those 657 for which WALDMEISTER can find a proof in 5 minutes. Of these, 190 specifications contain at least one AC-operator and 119 specifications contain at least one ACI-operator such that the corresponding criteria based on ground convergent subsystems are applicable. The specifications containing at least one AC-operator nearly all belong to one of the following five problem domains: (i) Robbins algebra, (ii) nonassociative rings, (iii) lattices, (iv) rings where $x^n = x$ implies commutativity, and (v) lattice-ordered groups. The specifications containing at least one ACI-operator belong (with one exception) to problem domains (iii) and (v). As general settings we use the autonomous mode of WALDMEISTER [HJL99] and vary the settings for the use of redundancy criteria. With WM we refer to the system that uses none of the criteria. It serves as a baseline in the experiments. In general, we used a timeout of 600 seconds for the experiments. Exceptions are given explicitly. This value is more than twice of the maximal time needed by WM.

Ground convergent subsystem for AC

We first consider four different variants of the criterion based on the ground convergent subsystem for AC of Section 6.2. As it turns out, two small variations can make a big difference in the prover's behavior. The first is the decision whether to keep redundant equations for simplification or to delete them. The second concerns equations (M) $x + (y + z) = z + (y + x)$ and (S) $x + (y + z) = y + (z + x)$. Both are redundant with respect to the criterion, but only by making C' "distinguished" (see p. 116), a somewhat arbitrary decision. For the second variation we make the distinction whether to treat (M) and (S) as usual or to protect them against the criterion and to use them for simplification and generation of critical pairs.

Table 6.1 contains experimental data for the four variants and the baseline system WM. In part (a) we give the running times in seconds for five individual examples, each of which is representative for one of the previously mentioned problem domains: GRP177-2 is a problem in the (v) domain; LAT020-1 a distributivity property for quasi-lattices (iii); RNG027-5 the left Moufang identity for (ii); RNG035-7 the instance of (iv) where $n = 4$; and ROB006-1 the theorem that the Winker condition $\exists x, y : x + y = y$ makes a Robbins algebra (i) Boolean. We can observe significant improvements with considerable variations between the different variants and the different examples. Considering the total times needed by all AC-problems, we see that deleting redundant

Table 6.1: Running times for different variants of the criterion based on AC-equality

	WM	no protection		protect (M) and (S)	
	(no criterion)	keep	delete	keep	delete
(a) running time in seconds					
GRP177-2	5.034	5.090	2.977	5.234	2.965
LAT020-1	173.898	64.587	70.526	67.678	49.030
RNG028-5	75.400	44.925	6.309	53.484	5.135
RNG035-7	9.082	6.460	8.813	6.559	6.557
ROB006-1	208.283	36.346	51.067	35.797	38.580
all AC-problems	2 115.415	1 092.137	757.849	1 363.059	580.409
(b) geometric mean of relative performance ($time_{\mathrm{WM}}/time_{\mathrm{variant}}$)					
all AC-problems	1.0000	1.1124	1.1929	1.0732	1.2310
$time_{\mathrm{WM}} > 0.5\,\mathrm{s}$	1.0000	1.3904	1.8132	1.2658	2.0054
$time_{\mathrm{WM}} > 5.0\,\mathrm{s}$	1.0000	1.7144	3.2613	1.5269	3.7660

equations is preferable to keeping them and that (M) and (S) should be protected. However, by considering plain running times the longer running examples dominate the analysis.

To compare the improvements independent of the running times we normalize them. The *relative performance* is the ratio of time needed by WM and the time needed by the specific variant. In part (b) of Table 6.1 we give the geometric mean of the relative performances which is appropriate for using normalized data [FW86]. The three rows contain the means for all examples, for the examples for which WM needs more than 0.5 seconds, and for the examples for which WM needs more than 5 seconds. We can observe for all variants that the longer the problems take the more they profit from the criterion. Considering the variants, we come for all chosen subsets to the same conclusions: Deleting redundant equations while protecting (M) and (S) gives the best improvements, followed by deleting redundant equations without giving (M) and (S) special treatment. Keeping redundant equations still gives improvements on WM, but then (M) and (S) should be treated normally. In the following, we will use WM-AC to refer to the variant of the last column (i. e., deleting AC-redundant equations while protecting (M) and (S)).

As already mentioned, the relative performance increases for longer-running examples. Concerning the used heuristics for selecting the next element of $\mathfrak{P}$, heuristics treating orientable and unorientable equations identical profit more than heuristics giving orientable equations preference.[11] Both observations can be explained by the data contained in Table 6.2 which presents numbers collected during the completion of the AC-subsystem with WM. To represent permutations of size n WALDMEISTER

[11] Note that the autonomous mode of WALDMEISTER usually prefers orientable equations, which applies to all examples containing AC-operators.

Table 6.2: The completion of the AC-subsystem in WM.

size of permutation	2	3	4	5	6	7
number of permutations	2	6	24	120	720	5 040
number of equations in WM	1	3	11	53	313	2 182
number of critical pairs in WM	1	20	369	9 371	331 722	15 888 156

needs about $n!/2$ equations, as it employs symmetries and smaller equations to represent all permutations of this size. With the criterion based on the ground convergent subsystem for AC, this number is reduced to overall three equations (or five when (M) and (S) are protected). This saves not only space but also a lot of calls to the ordering because of attempted ordered rewrite steps. But the real culprit lies in the number of critical pairs generated. As the last row shows, this number is for each level orders of magnitudes higher than $n!$. This illustrates the high redundancy that is present in the completion of the AC-subsystem itself[12]. All these critical pairs are avoided by the use of the criterion. The effect is further amplified by other equations in the completion. The permutation equations are unorientable, lead to many unifiers, and therefore produce many critical pairs. This explains why despite its simplicity and cheapness the criterion is highly beneficial in practice – when applicable. For challenging problems containing AC it is simply indispensable.

Ground convergent subsystem for ACI

For evaluating the criterion based on the ground convergent subsystem for ACI, we use as basis the variant WM-AC. Otherwise, the effects of the AC-subsystem concerning keeping or deleting redundant equations would dominate. In Table 6.3 we give the results for WM, WM-AC, and WM-AC using additionally ACI-equality, either keeping ACI-equal equations or deleting them. For each of the two problem domains, we give two examples. Furthermore, we give the total time needed as well as the geometric mean of the relative performance, in analogy to Table 6.1 for all examples and restricted to two subsets.

As we can see it is beneficial to keep equations that are ACI-equal, but not AC-equal. Deleting ACI-equal equations can even lead to a considerable slowdown of the prover. This differs strongly from the AC-case, where deleting equations is superior to keeping them. We can explain this as follows: ACI-equal equations may be useful as rules for simplifying critical pairs. As they are length-reducing, this influences the heuristic evaluation of critical pairs and hence the whole proof search. Compared to the impressive improvements of WM-AC over WM the additional improvements of using the ACI-redundancy criterion are rather small.

[12] Note that WALDMEISTER uses critical pair criteria based on symmetries: For example, it overlaps with the commutativity axiom only in one direction. Overlapping in the other direction would lead to critical pairs that are identical up to a variable renaming. Without these criteria, WALDMEISTER would generate even more critical pairs.

Table 6.3: Running times for WM, WM-AC, and two variants of WM-AC using the criterion based on ACI-equality

	WM	WM-AC	keep	delete
(a) running time in seconds				
GRP177-2	5.034	2.965	2.977	2.928
GRP181-1	5.250	4.688	4.631	4.579
LAT020-1	173.898	49.030	45.922	242.153
LAT044-1	67.089	38.734	36.194	50.175
all ACI-problems	532.250	366.469	354.222	746.126
(b) geometric mean of relative performance ($time_{\mathrm{WM}}/time_{\mathrm{variant}}$)				
all ACI-problems	1.0000	0.9969	1.0155	0.9387
$time_{\mathrm{WM}} > 0.5\,\mathrm{s}$	1.0000	1.0652	1.1269	0.8239
$time_{\mathrm{WM}} > 5.0\,\mathrm{s}$	1.0000	1.3094	1.3629	0.9194

Case splitting with variable constraints

The ground joinability test based on variable constraints is not limited to any specific theory; it is generic in nature. Nevertheless, in our experiments most examples that show improvements contain at least one AC-operator. There are only three other examples that profit noticeably from this criterion (BOO067-1, BOO076-1, and LAT070-1). It seems that in absence of AC-operators usually the number of redundant equations that can be detected by this criterion is rather small. Table 6.4 gives experimental data for WM, WM-AC, and WM-AC using additionally this ground joinability criterion, either keeping or deleting equations redundant by this criterion. It turns out that deleting redundant equations is not a good choice. Whereas some examples profit, the majority suffers from considerable slow-downs and several timeouts occur.

In contrast, keeping redundant equations leads in general to speed-ups, especially when AC-operators play a dominant role (such as in LAT020-1 or ROB006-1). As with the other criteria the longer the examples run the more they profit. However, several examples suffer a significant slowdown. With one exception they all belong to the domain of nonassociative rings. Also most of the timeouts of the variant that deletes redundant equations belong to this domain. It seems that in this domain the use of the criterion disturbs the proof search. In the preparation of the autonomous mode for WALDMEISTER [HJL99] we frequently observed that settings can lead to quite different results in different domains whereas within domains they have similar effects. As soon as the setting influences the proof search uniform behavior occurs quite rarely.

An interesting question is why deleting redundant equations is so significantly inferior? One reason is that the simplification relation is weakened by deleting redundant equations. As this is the main redundancy elimination mechanism the running time increases. Equations that could be shown redundant by a simple joinability test now have to be eliminated via the far more expensive ground joinability test. Analyzing the

Table 6.4: Running times for WM, WM-AC, and two variants of WM-AC using the ground joinability criterion based on variable constraints

	WM	WM-AC	keep	delete
	(a) running time in seconds			
GRP177-2	4.967	2.846	3.583	3.084
LAT020-1	171.161	47.831	35.737	> 600
RNG028-5	75.993	5.107	29.145	> 600
RNG035-7	8.982	6.463	6.050	> 600
ROB006-1	202.869	37.459	31.048	38.511
all AC-problems	2 093.715	566.914	683.965	17 247.748
# timeouts	0	0	0	28
all non-AC-problems	616.612	484.211	373.102	2 399.415
# timeouts	0	0	0	2
	(b) geometric mean of relative performance ($time_{\mathrm{WM}}/time_{\mathrm{variant}}$)			
all AC-problems	1.0000	1.2485	1.0497	0.5149
$time_{\mathrm{WM}} > 0.5\,\mathrm{s}$	1.0000	2.0236	1.6307	0.1913
$time_{\mathrm{WM}} > 5.0\,\mathrm{s}$	1.0000	3.9129	2.7493	0.2678
all non-AC-problems	1.0000	1.0024	0.8929	0.8880
$time_{\mathrm{WM}} > 0.5\,\mathrm{s}$	1.0000	1.0257	1.0592	0.5726
$time_{\mathrm{WM}} > 5.0\,\mathrm{s}$	1.0000	1.0442	1.1332	0.7109

performed ground joinability tests shows that then many of them are repeated because equations may be delivered several times after normalization. Keeping redundant equations avoids this and provides some kind of memory of successful ground joinability proofs. A second effect concerns the proof search. As the heuristic assessment of the critical pairs happens after normalization any depleting of the simplification relation can have a significant impact.

To analyze the influence of the subset heuristics and the use of subsumption steps, for each test we collected the number of cases actually needed. Table 6.5 contains the average number of cases grouped according to the number of different variables in the equation and whether the test was successful or not. We can see that the failure of the test can usually be detected with few cases. For successful tests the subset heuristics considerably reduces the number of cases, especially when subsumption steps are used. Furthermore, their use increases the success rate of the test from 6.4 % to 7.8 %.

In the following, we will use WM-GJ to refer to the version of WM-AC that uses the ground joinability criterion based on variable constraints and keeps redundant equations for simplification.

Table 6.5: Number of cases considered in the ground joinability criterion based on variable constraints

number of variables	2	3	4	5
maximal number of cases	3	13	75	541
(a) average number of cases without subsumption				
successful tests	3.0	6.2	16.5	156.4
failed tests	1.2	1.8	3.0	2.7
(b) average number of cases with subsumption				
successful tests	3.0	5.8	10.9	55.1
failed tests	1.2	1.8	2.9	2.4

Confluence trees

A problem concerning the evaluation of the ground joinability test based on confluence trees is that we currently have only an implementation available that is suitable for LPO. For many problems in TPTP the autonomous mode of Waldmeister selects a KBO as ordering. Hence, for evaluating confluence trees we have modified the settings to use always LPO.

As it turns out, confluence trees are not only the redundancy criterion with the most complex theoretical foundations, they are also the most difficult to implement and the most difficult to evaluate experimentally. Small variations in the implementation can result in large differences in the runtime behavior.

We have implemented five variants of confluence trees that we want to evaluate:

- $\textsc{ct}_1$ prefers rewrite steps to decomposition steps. It does not distinguish between full rewrite steps and constrained rewrite steps.
- $\textsc{ct}_2$ prefers decomposition steps to rewrite steps. It does not distinguish between full rewrite steps and constrained rewrite steps.
- $\textsc{ct}_3$ performs first full rewrite steps, then constrained rewrite steps, and finally decomposition steps.
- $\textsc{ct}_4$ performs first decomposition steps, then full rewrite steps, and finally constrained rewrite steps.
- $\textsc{ct}_5$ performs first full rewrite steps, then decomposition steps, and finally constrained rewrite steps.

In general, the use of confluence trees as a redundancy criterion results in a very unstable behavior of the prover. We were unable to distinguish groups of problems with similar behavior. The individual entries depicted in Table 6.6 (a) are chosen to illustrate this diversity. This is in clear contrast to the other criteria evaluated in this chapter which show usually a relatively uniform behavior, if not for all problems then at least for problem domains. It seems that the use of confluence trees has a stronger influence on the proof search than the use of the other criteria. Table 6.6 (a) does not

Table 6.6: Running times for WM, WM-AC, WM-GJ, and five variants of WM-AC using confluence trees

	WM	WM-AC	WM-GJ	CT_1	CT_2	CT_3	CT_4	CT_5
(a) running time in seconds (keeping redundant equations, nodes unlimited)								
BOO023-1	>600	>600	>600	10.94	6.92	10.31	5.89	6.28
BOO026-1	2.66	2.68	2.68	144.85	156.95	143.56	144.20	162.87
GRP181-3	200.53	190.74	127.84	104.57	153.51	94.08	92.88	87.69
GRP409-1	1.92	1.93	1.96	81.42	2.25	81.44	2.22	2.21
GRP547-1	9.49	9.43	>600	97.80	>600	28.75	>600	>600
LAT019-1	28.72	55.67	5.64	471.99	>600	201.28	4.86	4.97
RNG028-5	76.37	5.11	13.87	>600	>600	>600	68.65	68.20
ROB006-1	64.87	65.44	44.91	44.55	37.93	40.30	34.95	34.81
(b) geometric mean of relative performance ($time_{\text{WM}}/time_{\text{variant}}$) for $time_{\text{WM}} > 5.0$ s								
keep redundant equations								
unlimited	1.000	1.493	1.314	0.449	0.691	0.492	1.163	1.165
≤ 100 nodes	1.000	1.493	1.314	0.508	1.286	0.604	1.344	1.483
delete redundant equations								
unlimited	1.000	1.493	1.023	0.485	0.816	0.617	1.255	1.322
≤ 100 nodes	1.000	1.493	1.023	0.495	0.938	0.393	0.724	0.740

contain the accumulated times of the different variants because the many timeouts do not allow to draw any sensible conclusions from them.

Although we can observe that deleting redundant equations leads to repeated tests[13], deleting redundant equations is not per se inferior to keeping them. This is an interesting difference to the test based on variable constraints. Limiting the number of nodes in a tree improves performance of the tests, as can be gathered from Table 6.6 (b). However, this worsens the performance of CT_3, CT_4, and CT_5 when redundant equations are deleted. As already noted, problems can show a very individual behavior. Nevertheless, a general observation is that variants CT_4 and CT_5 show a clearly better performance than CT_1 or CT_3. If the number of nodes is not limited this holds also true for variant CT_2.

The overall irregular behavior of confluence trees makes it difficult to give recommendations for their usage. Depending on the problem, we can see significant speed-ups or considerable slow-downs. They can be a valuable tool for otherwise very difficult problems, but their behavior does not suggest automatic activation. Here, clearly the user is called upon to give them a try. For the rest of the experimental evaluation we therefore concentrate on "inner" aspects of confluence trees, namely, detection strength and time requirements of the different variants and the potential of the different optimizations discussed in Section 6.4.

[13] This is because equations are delivered several times after normalization. We could circumvent this by introducing a table storing the successful tests.

To evaluate detection strength and time requirements of the different variants we performed the following experiment (cf. Table 6.7): For each invocation of the ground joinability test, we ran the test based on variable constraints (column WM-GJ) and the five different variants based on confluence trees (columns $\text{CT}_1, \ldots, \text{CT}_5$). For each test, we recorded in addition to the result the number of generated nodes and the time needed. To get a reasonable approximation whether an equation is ground joinable (this is an undecidable property, see Theorem 3.4 on p. 26) we used the following approach: If none of the tests could show ground joinability we ran a version based on confluence trees that uses full backtracking (column CT^*). An equation is considered ground joinable if at least one of the tests showed ground joinability. To get consistent behavior during the proof, this value is then returned as result of the combined test. To avoid excessive running times, we set the maximal limit of nodes to 10000 and the maximal running time per problem to 1800 seconds. The maximal limit of nodes was mostly hit by the variant using full backtracking which needed most of the overall running time. Table 6.7 displays for each variant the number of successful ground joinability proofs, the average number of nodes per test, and the accumulated time in seconds.

The two variants CT_1 and CT_2, which do not have a special treatment of full rewrite steps, show an interesting complementarity: Although CT_2 needs many more nodes on the average than CT_1 it needs much less time overall. The explanation is that the constraints occurring in CT_1 are much more complex than the constraints occurring in CT_2. Therefore, the time spent in the constraint solver varies considerably between both variants.

The use of full rewrite steps is an important optimization. Variants CT_3, CT_4, and CT_5 need much less time than their counterparts CT_1 and CT_2. They do not only need fewer nodes on the average, they also have a slightly higher detection strength. A further observation is that preferring decomposition steps to constrained rewrite steps leads to a much better running time requirement. This finding is consistent with the overall observations previously discussed. Interestingly, as the small differences between CT_4 and CT_5 show, the order between full rewrite steps and decomposition steps is not important.

The test using variable constraints has a weaker detection strength than the tests using confluence trees. On the other hand, it has the best time requirements of all tests. Test CT^* shows that there is still a gap in the detection strength achieved by any of the redundancy tests to this more elaborate approximation of ground joinability. However, the costs are prohibitive. This variant is only sensible as a baseline to judge the other variants.

To assess the improvements obtainable from knowledge transfer along the branches in the confluence trees we implemented the following experiment: For each of the five variants of confluence trees we performed runs over all problems considered. For each test we recorded the result and the time needed by the standard implementation and the improved implementation using knowledge transfer. For each variant Table 6.8 shows the number of tests, the number of successful ground joinability proofs, and the accumulated times for standard and improved implementation. The number of tests and the number of successful proofs varies between the variants because of timeouts

Table 6.7: Detection strength, average number of generated nodes per test, and time requirements in seconds of the different generic ground joinability tests

	WM-GJ	CT_1	CT_2	CT_3	CT_4	CT_5	CT^*
strength in %	5.18	7.02	6.75	7.19	7.62	7.64	8.71
average number of nodes	4.7	10.6	31.3	8.8	9.4	8.8	2 028.2
successful tests	11.0	51.7	151.9	36.5	39.2	32.4	29.1
failing tests	4.1	6.9	21.1	6.1	6.4	6.3	2 025.7
accumulated time in s	14.6	3 226.8	228.3	2 415.1	41.6	42.4	111 799.4

Table 6.8: Advantages of knowledge transfer along the branches of confluence trees: times in seconds

	CT_1	CT_2	CT_3	CT_4	CT_5
number of tests	102 099	116 094	109 509	129 887	129 602
successful tests	10 402	12 691	12 946	18 787	18 750
time standard	7 850.61	2 071.82	8 339.92	911.42	872.95
time improved	2 626.86	2 039.66	2 196.13	786.56	683.80

and different time requirements. The advantages of using knowledge transfer differ considerably between the variants, CT_3 gains a speed-up of nearly four, CT_1 of about three. For CT_4 and CT_5 the speed-up is in the range of 15 to 30 per cent, CT_2 shows nearly no speed-up. Clearly, the variants that prefer constrained rewrite steps to decomposition steps show greater profit than the other variants which perform decomposition steps before constrained rewrite steps. The avoidance of unnecessary constraint satisfiability tests is far more important than the avoidance of other unnecessary operations.

The advantages of enriching the leave tests by subsumption and AC-equality can be derived from Table 6.9, which contains the data of CT_3 and CT_4.[14] Enriching the leave tests does not only reduce the average number of nodes but also increases detection strength. Additionally, the time requirements are reduced considerably. An interesting difference between variants CT_3 and CT_4 can be observed if we differentiate the average number of nodes for successful and for failing tests. Whereas in variant CT_3 this number is mostly reduced in the successful tests, in variant CT_4 the improvements are independent of the outcome of the test.

For limiting the branching factor of decomposition steps, we have implemented two optimizations: The first drops solved forms that are redundant, that is, all their covered ground instances are also covered by some other solved form. The second merges two solved forms if possible. Table 6.10 contains the corresponding data of variants CT_2, CT_3, and CT_4.[15] As can be seen, dropping redundant solved forms decreases the

[14] The data of variant CT_1 and CT_2 are similar to the data of CT_3 and the data of variant CT_5 are very similar to the data of CT_4. The experiment was similar to the previous one: For each invocation the test was run thrice with different parameter settings.

[15] The data of variant CT_1 are similar to the data CT_3 and the data of variant CT_5 are very similar to the data of CT_4.

Table 6.9: Advantages of using subsumption and AC-equality

	CT$_3$			CT$_4$		
	NONE	SUBS.	BOTH	NONE	SUBS.	BOTH
strength in %	11.07	11.27	11.30	13.77	14.22	14.24
average number of nodes	25.4	18.6	15.3	54.0	36.9	33.2
successful tests	123.9	70.4	42.3	217.8	150.6	130.7
failing tests	11.7	10.6	10.6	24.0	15.5	14.6
accumulated time in s	4 956.9	3 937.1	3 129.2	1 763.6	1 021.6	751.6

Table 6.10: Influence of modifying disjunctive constraints by dropping redundant solved forms or merging solved forms

	CT$_2$			CT$_3$			CT$_4$		
	NONE	DROP	BOTH	NONE	DROP	BOTH	NONE	DROP	BOTH
strength in %	10.87	10.89	11.18	12.17	12.17	12.17	14.38	14.52	14.13
avg. no. of nodes	166.5	93.0	35.3	18.0	14.7	12.7	35.5	20.2	23.5
successful tests	894.3	641.0	184.4	55.6	49.0	44.1	128.7	68.4	87.2
failing tests	69.3	23.2	14.7	11.3	8.8	7.3	17.0	10.2	11.1
accum. time in s	2 512.2	1 417.4	694.5	3 097.5	3 319.0	4 055.6	954.0	530.2	819.0

average number of nodes. For variants CT$_2$ and CT$_4$ this results in decreased running time. Variant CT$_3$, however, needs more time. This corresponds to the different strategies of the variants: CT$_2$ and CT$_4$ prefer decomposition steps and therefor gain much from a decreased branching factor. CT$_3$ performs decomposition steps after full and constrained rewrite steps which cause most of the branching; the gains are smaller than the costs of the optimization.

Merging solved forms is only a win for variant CT$_2$, which profits most from a reduced number of branches. Although for variant CT$_3$ the average number of nodes decreases, the running time increases. This is consistent with the previous observation concerning the dropping of unnecessary solved forms. Making decomposition steps more expensive results in an increase in the time needed. For variant CT$_4$ both values increase which we did not expect. We therefore analyzed individual confluence trees of variant CT$_4$. We can observe that often the merging of constraints prevent further full rewrite steps possible by instantiation or strengthening of the ordering. Furthermore, the merging is often undone by a subsequent constrained rewrite step.

An interesting example for which all of the implemented variants show exponential behavior on the average is the following. Let $\succ$ be the LPO for $f >_{\mathcal{F}} + >_{\mathcal{F}} a$, R contain $f(x,x) \rightarrow a$ and A, and E consist of $\mathsf{CC'}$. Consider for each permutation π the ground joinability proof of $f(x_1 + \cdots + x_n, x_{\pi(1)} + \cdots + x_{\pi(n)}) = a$. Then all implemented variants produce confluence trees for which the average number of leaves is exponential in n. This gives a lower bound on the running times as the number of

nodes is therefore on the average exponential in n as well. In the worst case, depending on n the number of leaves is $1, 1, 3, 13, 75, 541, 4683, \ldots$ for all variants. That is, all variants make in the worst case the same number of case distinctions like the method based on variable constraints (cf. p. 118). This indicates that all total quasi-orderings between $x_1, \ldots, x_n$ have to be considered. Of course, these problems are especially constructed, but similar problems occur often in AC-specifications. So the example gives at least some indication why sometimes so many nodes are considered especially in successful ground joinability proofs.

All in all, we can say that the use of confluence trees does unfortunately not lead to a predictable behavior of the prover. Here, further work is necessary. The main cause for their costs is not the use of a constraint solver which may potentially need exponential time. Rather it is the sheer number of nodes the trees may contain. Nevertheless, the use of an efficient constraint solver is essential because most of the time is spent analyzing the satisfiability of constraints. A limit on the number of nodes is a simple measure to prevent excessive runtime requirements. Furthermore, the different optimizations that we developed lead to considerable improvements. Not only do they increase detection strength but their combination can easily lead to a speed-up of more than one order of magnitude. This is important to turn confluence trees into a redundancy criterion that is practically feasible.

Ground reducibility

For analyzing the ground reducibility approach we use the four variants based either on WM-AC or on WM-GJ using either the standard (ACG) or the refined (ACG$^+$) version of the criterion for the AC-case. The variants based on WM-AC are called WM-ACG and WM-ACG$^+$, the variants based on WM-GJ are called WM-GJ/ACG and WM-GJ/ACG$^+$. For space reasons we omit WM in Table 6.11 and use WM-AC and WM-GJ as baselines instead. From the usual test-set, two problems show a peculiar behavior. LAT009-1 and LAT020-1 suffer a timeout whereas all other problems show improvements for ACG, and ACG$^+$ typically leads to further speed-ups. Analyzing the proof logs shows that in both problems critical pairs of ground reducible equations play an important role. For example, in LAT009-1 the activation of such an equation triggers a cascade of interreductions; the successful end of the proof search follows right after; the equation and some of its descendants occur near the end of the proof object. This is a typical phenomenon with redundancy criteria. A redundant equation which is not strictly necessary for finding a proof may nevertheless lead to significantly shorter proof searches.

For analyzing the effects of the criteria independent of the two problems with timeouts, Table 6.11 contains therefore additional entries presenting the data excluding both timeouts. We can see that ACG and especially ACG$^+$ show good improvements over the corresponding baseline versions. The improvements are relatively uniform (with the mentioned exceptions) and there is no problem domain (like nonassociative rings for WM-GJ) that shows a completely different behavior. The differences between WM-ACG and WM-GJ/ACG and similar WM-ACG$^+$ and WM-GJ/ACG$^+$ seem mainly inherited from the corresponding baseline versions. The criterion based on AC-ground reducibility detects most of the equations that the method based on ground joinabil-

Table 6.11: Running times for WM-AC, WM-GJ, and four variants using ground reducibility criteria

	WM-AC	WM-GJ	WM-ACG	WM-ACG+	WM-GJ/ACG	WM-GJ/ACG+
(a) running time in seconds						
GRP177-2	2.965	3.583	2.893	2.926	3.110	3.149
LAT020-1	49.030	35.737	> 600	> 600	> 600	> 600
RNG028-5	5.135	29.145	4.640	4.624	24.308	18.442
RNG035-7	6.557	6.050	6.062	4.901	5.388	5.107
ROB006-1	38.580	31.048	20.778	13.970	20.242	15.402
all AC-problems	580.409	683.965	1 605.374	1 525.069	2 107.483	1 757.052
w/o timeouts	521.229	642.692	405.133	324.659	906.979	556.558
# timeouts	0	0	2	2	2	2
(b) geometric mean of relative performance ($time_{\mathrm{WM}}/time_{\mathrm{variant}}$)						
all AC-problems	1.0000	0.8768	1.0560	1.0988	0.9358	0.9719
$time_{\mathrm{WM}} > 0.5\,\mathrm{s}$	1.0000	0.8711	1.0533	1.1941	0.8059	0.9034
$time_{\mathrm{WM}} > 5.0\,\mathrm{s}$	1.0000	0.7229	0.9438	1.0990	0.5036	0.6344
w/o timeouts	1.0000	0.8713	1.0943	1.1391	0.9685	1.0063
$time_{\mathrm{WM}} > 0.5\,\mathrm{s}$	1.0000	0.8524	1.1894	1.3547	0.9013	1.0145
$time_{\mathrm{WM}} > 5.0\,\mathrm{s}$	1.0000	0.6751	1.2503	1.4754	0.6316	0.8119

ity detects and needs much less time to do so. It even detects additional equations, because it uses full ordering constraints. Its purely forward approach allows to take more profit.

For the refined criterion ACG$^+$ we have to search for equations that cover the irreducible ground instances. In the standard test problems, the refined criterion was applicable in 14 578 cases (i. e., constraint Γ was determined satisfiable whereas constraint $\hat{\Gamma}$ was determined unsatisfiable). Of these in 194 cases the determined substitution bound no variable, hence no search was started. Of the 14 384 started searches none failed to determine a covering equation. To do so, very few overlaps are usually necessary, only 3.8 overlaps on the average. However, there are two outliers needing several hundred overlaps and one needs nearly 8 000. This suggests to introduce some limit and abandon the search when the limit is reached.

The real potential of the criterion based on ground reducibility can be seen in Table 6.12. It contains the runtime data for four challenging proof tasks.[16] Especially the refined criterion shows considerable improvements. For LAT018-1, WALDMEISTER has not yet found a proof without the use of the refined criterion. The only other prover we are aware of that has solved ROB007-1 and LAT018-1 is McCune's EQP [McC05b],

[16] Note that the RNG035-7 runs were performed on different hardware, namely, machines equipped with 2.6 GHz Xeon P4 and 2 GByte of memory.

Table 6.12: Running times in hours for challenging examples.

problem	WM-AC	WM-GJ	WM-ACG	WM-ACG$^+$	WM-GJ/ACG	WM-GJ/ACG$^+$
ROB020-1	7.5	6.0	3.3	2.6	3.1	2.6
ROB007-1	61.9	39.4	20.7	13.3	17.7	13.4
LAT018-1	>300	>300	>300	12.6	>300	13.2
RNG036-7*	>500	341.6	>500	151.6	243.1	112.0

* performed on different hardware, for details see footnote 16.

a system with built-in AC-theory. To our knowledge it needs highly unfair strategies to solve both problems. For example, it uses the super-0 strategy which discards most of the AC-unifiers. In contrast, the experiments of Table 6.12 were performed with fair strategies.

6.7 Related work and concluding remarks

The notion of redundancy was originally introduced to justify the usual simplification and subsumption methods in the superposition calculus [BG94]. We are only aware of very few publications aiming explicitly at the development of new redundancy criteria. One example is [LS95] where the authors develop redundancy criteria for constrained based completion techniques.

Of course, much work has been devoted to develop methods that avoid the generation of facts. In the unit equality case these are critical pair criteria. All constraint-inheritance based variants concentrate on avoiding the generation of redundant facts. Critical pair criteria are useful when working modulo AC, as then the simplification relation is expensive and many of the critical pairs caused by the prolific unification can be shown redundant [ZK90]. However, when working without builtin theory simplification is relatively cheap because of indexing techniques. Hence, in our experience the use of critical pair criteria does usually not improve the performance of a prover, actually, it leads often to slow-downs. One reason is that a critical pair criterion cannot prevent an equation from entering the completion process as different critical pairs may result in syntactical identical equations after simplification.

For several subsumption and simplification techniques versions that work on the ground level have been developed. H-subsumption [FL96] considers (clause) subsumption with respect to a fixed Herbrand-universe. Its main application domain seems to be automated model construction for which it is considered to be indispensable [EPW03]. We are unaware of an implementation in a high-performance theorem prover. The use of ground joinability to obtain ground confluence was first investigated in [MN90] and used to get the decidability result in [CNNR03]. To our knowledge, we were the first who implemented successfully such techniques in high-performance systems making ground joinability profitable in practice [AL01, AHL03]. Ground reducibility is

usually studied in the context of ordinary rewriting. An important application is in inductive theorem proving (see e. g. [JK89]) and a close relationship exists to the notion of sufficient completeness in algebraic specification [KNZ87]. In [CJ97] it was shown that ground reducibility is EXPTIME-complete for ordinary rewriting. As already mentioned, for ordered rewriting it is undecidable [CNNR03]. A work that has some similarities to our approach in Section 6.5.1 in that it uses constraint techniques for the AC-case is [PZ97] which was only prototypically implemented. Our work (published in [Löc04a]), however, was the first showing that ordering constraint-based redundancy criteria can speed-up high-performance systems considerably, thus attacking successfully the open problem 11 of [Nie99].

* * *

Concerning the use of the redundancy criteria, we can clearly recommend to activate the criterion that is based on AC-equality. The use of the criterion based on AC-ground reducibility and of the criterion based on variable constraints is usually profitable, too. For confluence trees, a clear recommendation is difficult: there are examples that show considerable improvements and there are other examples that suffer from significant slow-downs. As it is difficult to predict whether an example belongs to the first or to the second group, we can only suggest to try both variants. Except for AC-equal equations, it is usually profitable to keep redundant equations: they strengthen the simplification relation – thereby avoiding a lot of subsequent redundancy tests – and improve the heuristic evaluation of critical pairs.

If one is interested in implementing some of the criteria, we strongly recommend to implement first the criterion based on AC-equality. Its implementation is really simple and, when applicable, its use is very profitable and indispensable for demanding AC-problems. Secondly, one should consider implementing the criterion based on AC-ground reducibility – especially if a constraint solver is available or of further use in the system. The criterion is very helpful for better coping with demanding AC-specifications. To get a generic criterion, one has to consider the criteria based on case splitting. We do not recommend to implement confluence trees. Their implementation is by far the most difficult of all criteria considered and it is unclear how to take profit in the general case. Despite our considerable development efforts we have not succeeded in gaining overall profits from their use. In special circumstances their use may be extremely useful, but we have not yet found a domain where they show to be indispensable. Therefore, we suggest to implement the criterion based on variable constraints for the generic case.

As the experiments clearly demonstrate, the time spent for implementing redundancy criteria is very well invested: the system can cope much better with demanding specifications. Especially for AC-specifications, their use can make the difference between “impossible to prove in reasonable time” and “being a benchmark problem where the prover excels”.

7 A prover architecture

In saturation-based theorem provers, the reasoning process consists of constructing the closure of an axiom set under inferences. As is well-known, this process tends to quickly fill the available memory unless preventive measures are employed. Theorem proving procedures are refutationally complete under the idealized assumption of unlimited resources. But implementations thereof are run on finite machines and therefore can only analyze a finite part of the infinite search space. Therefore, the prover architecture has to support the efficient management of the available resources (especially memory) without preventing the use of powerful heuristics.

Most saturating high-performance provers nowadays use a form of the *given-clause algorithm* [McC97a]. As pointed out in [Vor01], there are two mainstream variants, called the DISCOUNT loop and the OTTER loop, and a considerable interest in the advantages and disadvantages of these variants has developed. In Section 7.1 we describe in detail the main proof procedure used by WALDMEISTER, which is in its original form a DISCOUNT loop. Besides discussing the differences to the OTTER loop and other variations of the main loop and an improved handling of the conjectures, we analyze the main weakness of this approach: its memory requirement. In Section 7.2 we present our new loop design which features a sophisticated set-based compression scheme. The experimental evaluation shows that with this scheme the memory requirements of the prover are dramatically cut down with only a moderate influence on the running times (see Section 7.3). In Section 7.4 we describe further benefits of the new prover architecture. Proof objects can be generated without any overhead, and the compact representations paves the way to an easily implementable parallelization of the prover with modest communication requirements. We conclude this chapter with Section 7.5.

7.1 An inspection of the proof procedure

Today, most saturating provers use a variant of the *given clause algorithm* [McC97a, Chap. 5.5.1] for their main loop. As basis for our exposition we take WALDMEISTER's original procedure (see Algorithm 1) and discuss other variants later. The input consists of an equation set $\mathcal{E}$, the *axioms,* a ground equation $\mathcal{C}$, the *conjecture*[1], a ground reduction ordering $\succ$, and a *weight function* φ mapping equations to *weights.* The saturation is performed in a cycle working on a set $\mathfrak{A}$ of *active facts* that participate

[1] It is easy to extend the procedure to handle a whole set of conjectures. The prover then tries to simultaneously find a proof for each element. Requiring only one proof leads to a disjunctive combination of the conjectures, requiring all proofs leads to a conjunctive combination of the conjectures.

Algorithm 1 The proof procedure of WALDMEISTER

FUNCTION $\textsc{Waldmeister}(\mathcal{E}, \mathcal{C}, \succ, \varphi)$: BOOLEAN

1: $(\mathfrak{A}, \mathfrak{P}) := (\varnothing, \mathcal{E})$
2: **WHILE** $\neg\mathsf{trivial}(\mathcal{C}) \wedge \mathfrak{P} \neq \varnothing$ **DO**
3: $e := \min_{\varphi}(\mathfrak{P})$; $\mathfrak{P} := \mathfrak{P} - \{e\}$
4: **IF** $\neg\mathsf{orphan}(e)$ **THEN**
5: $e := \mathsf{Normalize}^{\succ}_{\mathfrak{A}}(e)$
6: **IF** $\neg\mathsf{redundant}(e)$ **THEN**
7: $(P, \mathfrak{A}) := \mathsf{Interred}^{\succ}(\mathfrak{A}, e)$
8: $\mathfrak{A} := \mathfrak{A} \cup \{\mathsf{Orient}^{\succ}(e)\}$
9: $C := \mathsf{CP}^{\succ}(e, \mathfrak{A})$
10: $\mathfrak{P} := \mathfrak{P} \cup \mathsf{Normalize}^{\succ}_{\mathfrak{A}}(P \cup C)$
11: $\mathcal{C} := \mathsf{Normalize}^{\succ}_{\mathfrak{A}}(\mathcal{C})$
12: **END**
13: **END**
14: **END**
15: **RETURN** $\mathsf{trivial}(\mathcal{C})$

in inferences and a set $\mathfrak{P}$ of *passive facts* waiting to become members of $\mathfrak{A}$. The weight function φ realizes the heuristic part of the search. We require φ to induce a total ordering on equations. It has to ensure that every passive fact becomes minimal at some point in time and hence is selected. This guarantees the fairness of the proof procedure. We discuss weight functions in more detail later on (see p. 162).

The active facts $\mathfrak{A}$ induce, via their orientable instances $\mathfrak{A}^{\succ}$, an ordered rewrite relation $\longrightarrow_{\mathfrak{A}^{\succ}}$ and thereby a normal form relation $\stackrel{!}{\longrightarrow}_{\mathfrak{A}^{\succ}}$. As ordered rewriting is inherently terminating, every term has at least one normal form. We hence stipulate a total *normal form function* $\mathsf{Normalize}^{\succ}_{\mathfrak{A}} \subseteq \stackrel{!}{\longrightarrow}_{\mathfrak{A}^{\succ}}$ that is deterministic for any parameter value of $\mathfrak{A}$ and $\succ$. Furthermore, an *interreduction function* $\mathsf{Interred}^{\succ}$ is needed that, given $\mathfrak{A}$ and a new equation e, returns the set P of those active facts that are e-reducible and the remainder of $\mathfrak{A}$. For efficiency reasons, usually the interreduction of right-hand sides of rules is performed destructively. So $\mathsf{Interred}^{\succ}$ keeps rules with reducible right-hand sides in $\mathfrak{A}$ and returns in P only the rules with reducible left-hand sides and the reducible equations. Finally, let $\mathsf{CP}^{\succ}(e, \mathfrak{A})$ compute the set of all critical pairs between the equation e and all equations in $\mathfrak{A}$.

We are now ready to present the proof procedure in detail, see Algorithm 1. Further functionality required is informally explained subsequently. (L. 1) Initially, $\mathfrak{A}$ is empty and $\mathfrak{P}$ contains the axioms. (L. 2) The saturation proceeds as long as (i) the conjecture has not become trivial (i. e., joined), and (ii) there are still passive facts. Inside the completion loop, the following steps are performed: (L. 3) Select an equation e from $\mathfrak{P}$ which has minimal weight with respect to φ. In the terminology of [McC97a] e is the given clause. (L. 4) Skip if a parent equation has been reduced meanwhile. It suffices to only consider critical pairs with persistent parents. Equation e has not participated actively in any simplifying inference and hence is redundant. (L. 5) Simplify e to a

normal form. (L. 6) Skip if e has become trivial now, or redundant in some other way, e. g. via subsumption. (L. 7) Interreduce the set $\mathfrak{A}$ with e and remove e-reducible active facts from $\mathfrak{A}$. (L. 8) Check whether e is orientable, tag it accordingly, and add it to $\mathfrak{A}$. (L. 9) Generate all new critical pairs between e and $\mathfrak{A}$. (L. 10) Normalize the generated critical pairs and the reducible active facts, and add the nonredundant ones to $\mathfrak{P}$. (L. 11) Normalize the conjecture.

For this proof procedure, the following two invariants hold: (i) Every nonredundant one-step conclusion of $\mathfrak{A}$ is contained in $\mathfrak{A} \cup \mathfrak{P}$. (ii) The active facts $\mathfrak{A}$ are completely interreduced. Hence, in case of termination with $\mathfrak{P} = \varnothing$, $\mathfrak{A}$ is an interreduced rewrite system, convergent on ground terms and equivalent to $\mathcal{E}$. Thus, it constitutes a decision procedure for the uniform word problem of $\mathcal{E}$. Furthermore, if φ ensures fairness, then the procedure proves any valid conjecture $\mathcal{C}$ within finite time. Because we initially insert the input axioms $\mathcal{E}$ in $\mathfrak{P}$ there is no need of a special input reduction as in [Wei01]. The same effect is achieved automatically by invariant (ii).

In Chapter 6 we distinguish two kinds of redundant equations: some are deleted, others are kept. By using two predicates, $\mathsf{redundant}_{\text{delete}}$ and $\mathsf{redundant}_{\text{keep}}$, we can modify Algorithm 1 accordingly. In line 6 predicate $\mathsf{redundant}$ is replaced by $\mathsf{redundant}_{\text{delete}}$; and line 9 is modified to:

$$\textbf{IF } \mathsf{redundant}_{\text{keep}}(e) \textbf{ THEN } C := \varnothing \textbf{ ELSE } C := \mathsf{CP}^{\succ}(e, \mathfrak{A}) \textbf{ END}$$

Furthermore, we mark equations e with $\mathsf{redundant}_{\text{keep}}(e) = \mathsf{TRUE}$ such that $\mathsf{CP}^{\succ}(e', \mathfrak{A})$ avoids them for computing critical pairs.

To get an abstract notion of time we count the iterations of the loop core in lines 7–11. We then use indices to refer to the values of variables at a certain time, more precisely, the values they have after the i-th iteration of the loop core. For example, $\mathfrak{A}_i$ denotes the value of $\mathfrak{A}$ at time i (i. e., including the i-th activated fact e_i, but without the elements that were interreduced by e_i), C_i denotes $\mathsf{CP}^{\succ}(e_i, \mathfrak{A}_i)$, etc. The sets $\mathfrak{A}_i$ induce the rewrite relations $\longrightarrow_i \; := \; \longrightarrow_{\mathfrak{A}_i^{\succ}}$ and the normal form functions $\mathsf{Norm}_i := \mathsf{Normalize}^{\succ}_{\mathfrak{A}_i}$.

The passive facts are, as can be seen in the proof procedure, subject to normalization only right after their generation and in case they get selected. This variant of given-clause algorithm is called *Discount loop*. In contrast, within an *Otter loop* the whole set of passive facts is normalized in every iteration of the cycle. Line 10 then reads

$$\mathfrak{P} := \mathsf{Normalize}^{\succ}_{\mathfrak{A}}(\mathfrak{P} \cup P \cup C)$$

instead. Some Otter loop implementations additionally use the rewrite relation generated by the union of active and passive facts, i. e. they always employ $\mathsf{Normalize}^{\succ}_{\mathfrak{A} \cup \mathfrak{P}}$ instead of $\mathsf{Normalize}^{\succ}_{\mathfrak{A}}$.[2] Then they keep as invariant that $\mathfrak{A} \cup \mathfrak{P}$ are completely interreduced.

The loop variants are named after well-known respective implementations, cf. [ADF95] and [McC05a]. An early experimental comparison of both loops was done in [Küc82,

[2] More precisely, this involves a more subtle form of interreduction. Note also that then the orphan predicate used in line 4 requires tagging passive facts that have ever participated in reductions, and that only untagged orphans may be deleted.

p. 131], where, based on observations when completing the group axioms using the ALDES Data-Type Completion System, preference is given to the OTTER loop. The loop names have been coined in [RV03, Sect. 2 and 6], where also extensive experiments of both variants are reported. A discussion within the frame of resolution and superposition is contained in [Wei01, Sect. 3 and 5.3].

As nothing is done with passive facts but selection, it is clear that the DISCOUNT loop leads to a simpler system design, because no term indexing is necessary on the elements of $\mathfrak{P}$ as would be the case if $\mathfrak{P}$ were to be kept fully normalized. Apart from relieving the system developers, this should also reduce the system run time, presuming that the search behavior of the prover is essentially the same. Admittedly this is an issue still under discussion. In [BH96] experiments with interreducing $\mathfrak{P}$ at certain intervals are documented. Such interreductions usually lead to small savings in abstract time needed to find a proof (up to 10 %). As this does not justify the costs, WALDMEISTER does not interreduce $\mathfrak{P}$.

There are several other variants of the main completion algorithm. They mostly differ in the way the set of critical pairs is computed. McCune [McC97a] discusses the *pair algorithm* as an alternative to the given-clause algorithm. In this variant, only the critical pairs belonging to a pair of active facts are computed at a time. Then a complete interreduction is performed before another pair of facts is selected. For the completion procedure in [Hue81], Huet discusses several variants of the way critical pairs are computed. The most radical one is to compute only one critical pair at a time and then perform a complete interreduction. Usual descriptions of Buchberger's algorithm to compute Gröbner-bases use a similar description, for example only one s-polynomial is considered at a time in [GCL92].

Common heuristics

The weight function provides the heuristic part of the search. Elements with smaller weights are preferred, that is, if $\varphi(e) > \varphi(e')$ then e' is considered to be "better" than e. To establish the fairness of the proof procedure the weight function has to ensure that every passive fact becomes minimal in $\mathfrak{P}$ at some time. The most simple way to achieve this is to order the passive facts according to their age: older facts are preferred to newer ones. This can be implemented for example by using timestamps. Because this leads to a first-in first-out behavior of $\mathfrak{P}$ we call the corresponding weight function φ_{fifo}. As weights do not change over time, weight-based heuristics cannot respond to important changes in the proof state (e. g., the occurrence of a certain important equation or the interreduction of most of the active facts). On the other hand, they lead to a simple implementation of $\mathfrak{P}$ with a priority queue.

In WALDMEISTER, the most frequently used weight functions are the following:

$$\begin{aligned}
\varphi_{\text{add}}(s = t) &= |s| + |t| \\
\varphi_{\text{gt}}(s = t) &= \begin{cases} |s| & \text{if } s \succ t \\ |t| & \text{if } t \succ s \\ |s| + |t| & \text{otherwise} \end{cases} \\
\varphi_{\text{mix}}(s = t) &= \varphi_{\text{add}}(s = t) \cdot \varphi_{\text{gt}}(s = t)
\end{aligned}$$

Function φ_{add} prefers equations with small terms. Smaller terms are usually better suited for simplification (fewer function symbols make matching easier) and provide fewer positions for overlapping. Function φ_{gt} concentrates more on the second aspect. If the equation can be oriented into some rule, it uses the length of the left-hand side. Otherwise, it adds the lengths of both sides, which corresponds to the fact that we have to overlap into both sides of an equation. Function φ_{mix} is a simple combination which is often superior to its ingredients. Nevertheless, there are examples that are easy exercises with one of the heuristics and take a long time with either of the others. By assigning individual weights to variables and function symbols the three basic weight functions can be generalized. Instead of using the length of a term its weight is then determined as the sum of the weights of the symbols it contains.

Note that the initial normalization of critical pairs in line 10 of Algorithm 1 is an essential prerequisite for these simple heuristics to work well. If omitted, the proof search takes much longer. The initial normalization helps to approximate the shape of the fact when they get activated. The deletion of critical pairs that is possible because they are joined is only a secondary effect.

One of our requirements is that the heuristics shall induce a total ordering on the passive facts. When several equations have the same weight ambiguities arise which we want to avoid. For example, a total ordering on $\mathfrak{P}$ provides stability to the behavior of the system against different implementations of the priority queue. Therefore, we usually refine weight functions that do not induce a total ordering (such as φ_{add}, φ_{gt}, and φ_{mix}) with φ_{fifo}. Thus, weights can be in general tuples of numbers that are compared lexicographically. As the length of theses tuples is fixed for a given heuristics, storing a weight needs constant space. Of the elements that have identical weight according to the main weight function the oldest one is then preferred.

An important extension are *multi-dimensional heuristics*. These use several weight functions $\varphi_1, \ldots, \varphi_k$. When the minimal element of $\mathfrak{P}$ is required they first select one of the weight functions (i. e., the dimension). Then they determine the minimal element according to this weight. Usually, a round-robin scheme is used to select the dimensions. A well-know instance of this approach is OTTER's pick-given ratio: After for example five clauses which are selected according to their weight, the oldest clause is chosen. Many provers use similar schemes, probably the prover E [Sch02] provides the most elaborate one. Multi-dimensional heuristics can be implemented using several priority queues. One then has to ensure that no element is selected repeatedly. A more elegant solution is the use of multi-dimensional heaps [DW93].

A set-based representation of the conjecture

A nuisance with our main-loop is that most time is spent on completing the input axiomatization. For the handling of the conjecture less then 1 % is invested. This has been criticized as lack of goal orientation. Certainly, proving the conjecture is our main aim. Therefore, we are willing to invest more time in handling the conjecture, provided this shortens the time to find a proof.

The conjecture $u = v$ is proved by joining u and v. However, the rewrite relation induced by $\mathfrak{A}$ is not ground confluent during the completion process. Hence, the normal

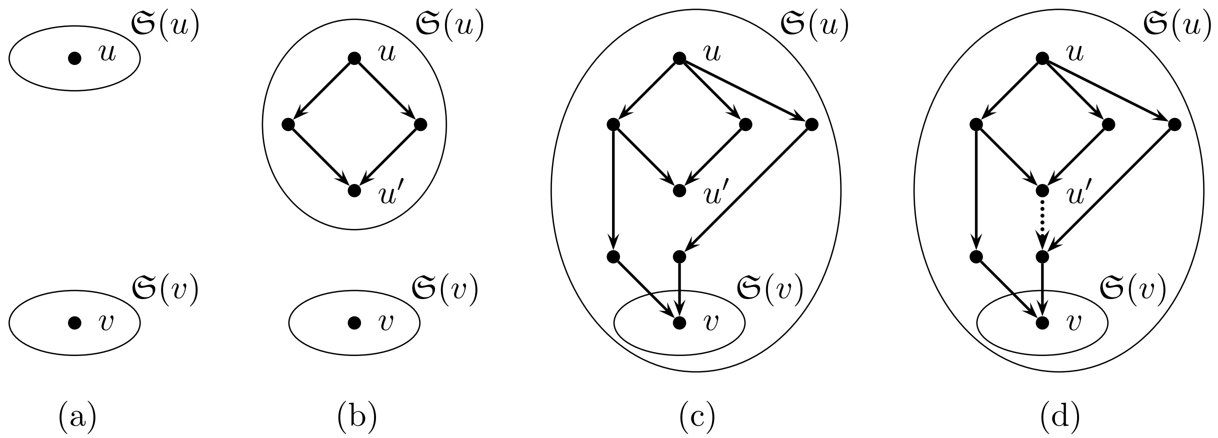


Figure 7.1: Sets of successors during the proof search for GRP141-1

forms of u and v are not necessarily unique. So it is possible to strengthen the usual joinability test by considering all rewrite successors of u and v. Instead of considering an arbitrary normal form, WALDMEISTER incrementally constructs for each term in the conjecture the set $\mathfrak{S}$ of all rewrite successors [Jae97]. This refines the functionality specified in line 10 of Algorithm 1. The conjecture $u = v$ is proved if $\mathfrak{S}(u) \cap \mathfrak{S}(v) \neq \emptyset$.

The method is best explained with a simple example. Figure 7.1 shows how sets of successors can be used in solving the proof problem GRP141-1. The terms $u \equiv f(f(a,b),c)$ and $v \equiv c$ have to be joined. The four graphs illustrate the enlarging of the conjecture. (a) Initially, the two sets consist of only one term each. (b) With the 16th rewrite rule u can be reduced to two different terms which join again in the term denoted by u'. $\mathfrak{S}(u)$ now consists of four terms. (c) After generating rule no. 19, u can be reduced to v in two different ways, hence $v \in \mathfrak{S}(u)$ and the conjecture is proved! Compare this with the standard approach which would still be stuck in u'. (d) The last graph shows the way out of this dead end which is not possible before 31 rules have been generated. The number of generated rules is reduced by one third in this example. In more difficult proof tasks, even higher reduction factors may occur.

This technique has considerably improved WALDMEISTER. When introduced, it increased the number of solvable TPTP problems from 278 to 300, and even decreased the average proof time. For the full superposition calculus, the Vampire prover features a similar technique under the name "negative equality splitting", which is inspired by ours [RV01]. We have also investigated the usefulness of this approach as redundancy criterion. However, it turned out to be too costly for that purpose. Details can be found in [Jae97].

Reducing the space requirements

Typically $\mathfrak{P}$ contains far more equations than $\mathfrak{A}$, as a rule of thumb $|\mathfrak{P}| = O(|\mathfrak{A}|^2)$ on the average (i. e., one can fit parabolas into the plot of these quantities). Therefore, the space requirements of WALDMEISTER are dominated by the representation of the passive facts. Hence, methods to save space mainly have to tackle $\mathfrak{P}$. From very early stages in the development [BH96, Buc97], we have used simple *compression schemes* to

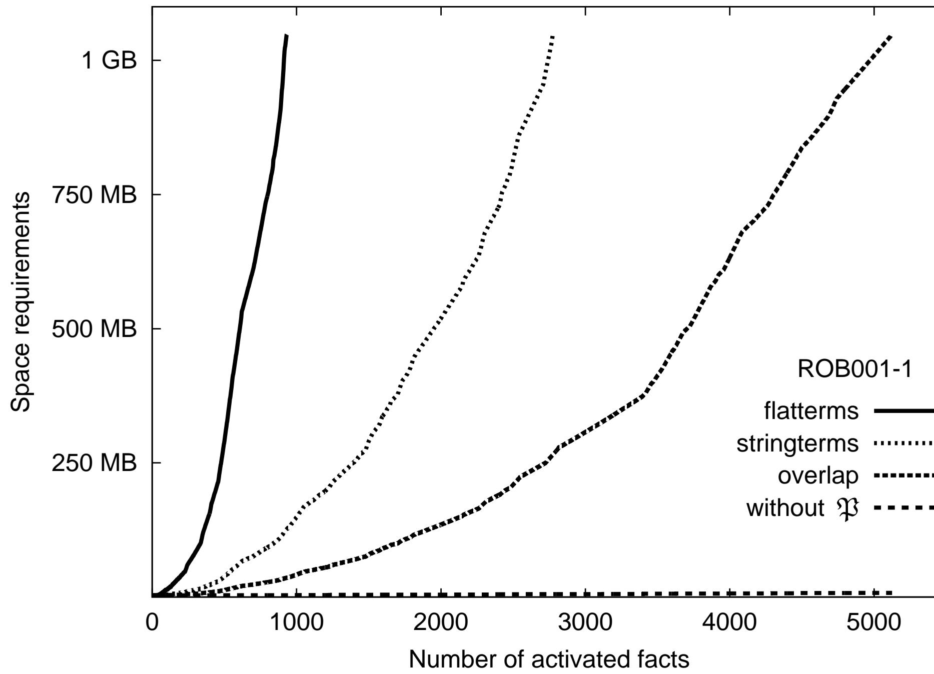


Figure 7.2: Size of the process over abstract time

minimize the space needed for single elements. The main motivation was the size of the memory the machines used for the development were equipped with. An appreciated side-effect were small reductions in running times, smaller process sizes fit better the memory hierarchy of the machines. Within Waldmeister, compression can be done in three ways: (i) Not at all, i. e. flat-terms are employed at the cost of 12 bytes per symbol. (ii) String-Terms: Terms are stored simply as strings of symbols, at the cost of one byte per symbol. (iii) Overlap representation: After heuristic assessment, the terms themselves are thrown away, and just minimal reconstruction information is kept, at constant cost.

The influence of these representations on the process size as a function of abstract time can be studied in Figure 7.2. For each variant, the Waldmeister system was run on the ROB001-1 proof task from the TPTP problem library [SS98] using an UltraSPARC-IIi workstation. The process size minus the space needed for $\mathfrak{P}$ serves as a baseline (lowest curve). String-Terms lead to compression rates of about 10, which is comparable to what can be achieved with a shared term representation for unit equational problems [LS01]. The overlap representation improves on this by another factor of nearly 10. But still here, the overall space requirements, exceeding 1 GByte after some five thousand activation steps, are mostly caused by the data structure for $\mathfrak{P}$, which mirrors the estimated quadratic relation between $|\mathfrak{A}|$ and $|\mathfrak{P}|$. Of course, one can equip the machines with more memory. But this does not solve the problem. For 32-bit processors, we quickly reach the limitations of address space. Furthermore, the space requirements have clearly quadratic asymptotic behavior. Hence, they soon will exceed the limits of any reasonable budget.

The overlap representation is apparently the best one in terms of space. But – contrary to one's expectations – it may influence the behavior of the proof search. The reason is that the rewrite relation induced by $\mathfrak{A}$ changes between the insertion of a passive fact at time i and its reconstruction at time j. Therefore, for the original critical pair $s = t$ the reconstructed equation $\mathsf{Norm}_j(s = t)$ may considerably differ from the equation originally inserted, namely $\mathsf{Norm}_i(s = t)$. Thus, the reconstruction is *inexact*. This is highly undesirable. The representation of passive facts is a completely internal matter of the subsystem implementing $\mathfrak{P}$. From a user's perspective, changing the representation should be completely neutral for the proof search. In terms of software engineering, this means that the I/O-behavior of the module implementing $\mathfrak{P}$ shall be independent of internal data structures: For any sequence of insertions and deletions of passive facts the module has to deliver exactly the same data as the implementation based on flat-terms would do.

7.2 The Waldmeister loop

As has become clear in the previous section, the main limitation of saturating provers are the memory requirements. Therefore, a variety of incomplete strategies has been employed for discarding formulas that are deemed to be unnecessary for the proof search (e. g., because they exceed a certain weight limit). In our opinion, such incomplete strategies are undesirable for at least three reasons: First, in laying the theoretical foundations the most effort is usually spent in showing the completeness of the calculus. This is simply sacrificed by using such incomplete schemes. Second, the user of a theorem prover has to be concerned about the weight limits in use. Is the limit too small, the prover may fail to find a proof. Is it too large, the prover may take too long or may need too much memory such that the proof attempt has to be abandoned. And thirdly, sometimes it is possible to reject an invalid conjecture. This is made impossible by discarding formulas. In the following we will present an approach that avoids all of the problems mentioned above and nevertheless keeps the memory requirements low.

As we have seen, the representation of $\mathfrak{P}$ requires most of the memory. To reduce these requirements, we therefore have to improve this representation. To do so, we develop further the idea of compression and reconstruction at need. This is possible because elements of $\mathfrak{P}$ are really passive; they do not participate in any inferences. The key idea of our approach is to represent *whole subsets* of $\mathfrak{P}$ by *single entries*. These subsets are not arbitrary, but reflect the structure of the given-clause algorithm. The first kind of sets contains the critical pairs belonging to a *p*air of equations e_i and e_j. We therefore call them *p-sets*. The second kind of sets, called *a-sets*, consists of all the critical pairs jointly generated at the *a*ctivation of one given clause. As these sets are structured they can be represented in the same compact way as individual entries. Especially, we will show a variant that guarantees that the size of the representation of $\mathfrak{P}$ is linear in abstract time.

For intrinsic completeness reasons, an important prerequisite to our technique is the availability of the *collected history of* $\mathfrak{A}$, i. e., all the preceding states $\mathfrak{A}_1, \mathfrak{A}_2, \ldots$ of the active facts, to re-induce the corresponding normalization functions Norm_i. An elegant and efficient realization will be discussed in Section 7.2.2.

7.2.1 A space-efficient representation of $\mathfrak{P}$

The task of the subsystem $\mathfrak{P}$ is to fairly enumerate all inserted equations with the help of a weight function φ. $\mathfrak{P}$ communicates with its environment via two methods: insert inserts all (nonredundant) critical pairs between an equation e_i and $\mathfrak{A}_i$ at once. Additionally, all equations interreduced by e_i are inserted. The other method is deleteMin. It extracts the minimal equation from $\mathfrak{P}$. Concerning the abstractly formulated Algorithm 1, deleteMin realizes the functionality for line 3, and insert is needed in line 10.

We require an implementation of $\mathfrak{P}$ to be neutral with respect to inferences and proof search. Even if it performs internally some inferences for reconstructions it shall appear from the outside like a mere container, namely as a priority queue storing the unmodified equations augmented with their weight computed by φ. (We discuss the modifications of our scheme that are necessary to support multi-dimensional heuristics later, see p. 172). Of course, this idealized *reference implementation* assumes the availability of unlimited memory. An actual implementation shall then show the same input/output (I/O)-behavior. That is, for the same sequence of insert and deleteMin operations (with the same arguments for insert), it shall return the same equations for deleteMin operations as the reference implementation does.

Reconstructing individual equations

In a first step, we only consider the use of compact representations for individual elements and show that, provided the collected history of $\mathfrak{A}$ is available, the required I/O-behavior can be achieved. Given two equations e_i and e_j, $i \geq j$, the critical pair at the overlap position p is uniquely determined. The position information can be encoded in a single integer, e. g. as number of the top symbol of the subterm identified by p when the symbols are counted from left to right. We extend p to an *extended position* xp with three more bits. These encode the sides that are overlapped and whether p is a position of e_i or e_j. If $s = t$ is the critical pair, its weight w is calculated as $w = \varphi(\mathsf{Norm}_i(s = t))$. The weighted critical pair can be represented within constant space via its *overlap entry* $\langle w, i, j, xp\rangle$. Equations stemming from interreduction can be encoded in a similar way: When e_j gets reduced by e_i, $i > j$, the corresponding *interreduction entry* is $\langle w, i, j, \dagger\rangle$, with $w = \varphi(\mathsf{Norm}_j(e_i))$ and a special symbol †. We use *i-entries* to refer to entries that describe individual passive facts, such as overlap or interreduction entries.

The data in the i-entries is sufficient for the exact reconstruction of the equations:

LEMMA 7.1 *Provided the collected history of $\mathfrak{A}$ is available, an implementation of $\mathfrak{P}$ using only i-entries shows the same I/O-behavior as the reference implementation.*

PROOF Let $s = t$ be the critical pair between equations e_i and e_j, $i \geq j$, at the (extended) position xp. When $s = t$ is inserted, $\mathfrak{P}$ internally stores the overlap entry $\langle w, i, j, xp\rangle$ where $w = \varphi(s' = t')$ with $s' = t' = \mathsf{Norm}_i(s = t)$. The reference implementation stores $\langle w, s' = t'\rangle$. When this entry is selected, function deleteMin has to

reconstruct $s' = t'$ from $\langle w, i, j, xp \rangle$. This can be accomplished in the following way: Equations e_i and e_j are contained in $\mathfrak{A}_i$. We can therefore reconstruct the overlap by unifying the subterms identified by xp. Furthermore, Norm_i is available in the collected history of $\mathfrak{A}$. Hence, we can compute $\mathsf{Norm}_i(s = t)$ as required for returning $s' = t'$.

With a similar approach we can also reconstruct interreduced equations. □

Compressing subsets of $\mathfrak{P}$

In the second step, we define entries that represent more than one equation. Assume that equations e_i and e_j, $i \geq j$, have n nonredundant critical pairs. To represent them we need n overlap entries $\langle w_1, i, j, xp_1 \rangle$, ..., $\langle w_n, i, j, xp_n \rangle$. They are selected with increasing weights. As these do not change over time, the relative ordering between the overlap entries belonging to one p-set is fixed. This means that at any point τ in time after the insertion of these entries, there exists some weight w_τ such that all entries $\langle w_k, i, j, xp_k \rangle$ with $w_k < w_\tau$ are already selected and all entries $\langle w_m, i, j, xp_m \rangle$ with $w_\tau \leq w_m$ are still in $\mathfrak{P}$. Hence, ignoring interreduction entries, $\mathfrak{P}$ can be considered as a priority queue that contains priority queues containing overlap entries.

Experience tells us that in the common case from each p-set only the (usually few) critical pairs with rather small weights are selected. The majority of critical pairs has larger weights and stays in $\mathfrak{P}$ very long. During the proof search many "better" critical pairs may be generated for other parent equations. So it is unlikely that the "heavier" critical pairs will ever be selected.

This leads to the main idea of our set-based compression approach. Given equations e_i and e_j, $i \geq j$, and a *weight limit* w, we can combine all overlap entries $\langle w_m, i, j, xp_m \rangle$ with $w \leq w_m$ into the single *p-entry* $\langle w, i, j, * \rangle$. The idea is formally captured in the following definition:

Definition 7.1 *A p-entry* $\langle w, i, j, * \rangle$ represents *the set of overlap entries* M *iff*

$$M = \big\{ \langle w', i, j, xp \rangle \mid s = t \in \mathrm{CP}(e_i, e_j) \text{ at extended position } xp, \\ s' = t' = \mathsf{Norm}_i(s = t),\ s' \not\equiv t',\ w' = \varphi(s' = t'),\text{ and } w \leq w' \big\} ,$$

it represents exactly M *if furthermore there is some* $\langle w', i, j, xp \rangle$ *in* M *with* $w = w'$.

By using p-entries, we gain a space reduction, because one p-entry represents several critical pairs and replaces the corresponding overlap entries. Note that p-entries generalize p-sets: they do not represent all critical pairs belonging to a pair of equations, they represent only the "heavier" ones with low chances to be selected. For the promising critical pairs that are likely to be selected we still use overlap entries and keep them in some kind of cache. Interreduction entries are always kept individual; they are never represented in a p-entry. To simplify the approach we use the convention that for any pair of equations at most one p-entry is contained in $\mathfrak{P}$.

Given a p-entry we can *decompress* it, that is, we can reconstruct all overlap entries it represents in an exact way:

LEMMA 7.2 *Provided the collected history of* $\mathfrak{A}$ *is available, the p-entry is of the form* $\langle w, i, j, *\rangle$*, and the current time is at least* i*, we can implement function* decompress *such that* $\mathsf{decompress}(\langle w, i, j, *\rangle) = M$ *iff* $\langle w, i, j, *\rangle$ *represents* M*.*

PROOF Equations e_i and e_j are contained in $\mathfrak{A}_i$ which is available by assumption. Then we can simply recompute all critical pairs between e_i and e_j. After normalizing them with Norm_i and discarding them in case they can be joined, we compute the corresponding weights. We then eliminate all critical pairs with weight strictly smaller than w. From the remaining ones we derive M. □

Note that it is crucial for this decompression of the p-entry that the normal form function Norm_i is available. Using a different normal form function Norm_j can have the following effect: Let $s' = t' = \mathsf{Norm}_i(s = t)$ and $s'' = t'' = \mathsf{Norm}_j(s = t)$. Because the normal form functions differ, $s' \equiv s''$ and $t' \equiv t''$ need not necessarily hold. If $\varphi(s'' = t'') \geq w > \varphi(s' = t')$ we erroneously insert an overlap entry into M. If $\varphi(s' = t') \geq w > \varphi(s'' = t'')$ we erroneously omit an overlap entry. This means that if we use p-entries for compression and such errors occur we cannot longer guarantee the correct I/O-behavior of $\mathfrak{P}$. The first case is relatively harmless, a critical pair might be delivered repeatedly. The second case, however, is very serious: we lose completeness! Hence, the availability of all normal form functions Norm_i is essential for our approach.

In our implementation we want to establish the following invariant: each critical pair is represented exactly once (i. e., as an overlap entry or it is contained in some p-entry). Hence, compression of overlap entries into p-entries has to be done properly. We say that replacing the set of overlap entries $M = \{\langle w_1, i, j, xp_1\rangle, \ldots, \langle w_k, i, j, xp_k\rangle\}$ by the p-entry $\langle w, i, j, *\rangle$ is a *valid compression* iff the p-entry represents exactly M. That is $w = \min\{w_1, \ldots, w_k\}$ and $\mathfrak{P} - M$ contains neither an overlap entry $\langle w', i, j, xp'\rangle$ with $w' \geq w$ nor a p-entry of the form $\langle w', i, j, *\rangle$. As a consequence, performing valid compressions preserves the invariant.

When the deleteMin operation is slightly modified, Lemma 7.1 carries over: Whenever a p-entry becomes minimal the appropriate decompression operation is invoked before deleting the minimal entry.

LEMMA 7.3 *Provided the collected history of* $\mathfrak{A}$ *is available, an implementation of* $\mathfrak{P}$ *using i-entries, p-entries, and only valid compressions shows the same I/O-behavior as the reference implementation.*

PROOF Let the reference implementation select $s = t$ which is either an interreduced equation or a critical pair. The first case is analogous to Lemma 7.1. For the second case, let $s = t$ be the critical pair between equations e_i and e_j at extended position xp with weight w. Each equation is represented exactly once in the compression based implementation. Hence, it is either represented with the overlap entry $\langle w, i, j, xp\rangle$ or it is contained in some p-entry. Again, the first case is analogous to Lemma 7.1. The second case remains. Recall that the ordering on weights is total and that the elements are selected in strictly increasing order. Furthermore, by the condition of a valid compression, all p-entries represent exactly their set of overlap entries (i. e., their weight

coincides with the weight of the minimal overlap entry they represent). Hence, the p-entry has the form $\langle w, i, j, *\rangle$ and is minimal in $\mathfrak{P}$. After replacing the p-entry with $\mathsf{decompress}(\langle w, i, j, *\rangle)$ the minimal element is an overlap entry, from which the equation can be reconstructed as in Lemma 7.1. □

By abstracting from the third component of an entry we can compress even more critical pairs into one entry. An *a-entry* $\langle w, i, *, *\rangle$ represents all critical pairs that were generated at the activation of e_i, that were nonredundant after simplification, and that have a weight assigned by φ that is greater than or equal to w.

Definition 7.2 *An a-entry* $\langle w, i, *, *\rangle$ represents *the set of overlap entries* M *iff*

$$M = \big\{\langle w', i, j, xp\rangle \mid e_j \in \mathfrak{A}_i,\ s = t \in \mathrm{CP}(e_i, e_j) \text{ at extended position } xp, \\ s' = t' = \mathsf{Norm}_i(s = t),\ s' \not\equiv t',\ w' = \varphi(s' = t'), \text{ and } w \le w'\big\}\ ,$$

it represents exactly M *if furthermore there is some* $\langle w', i, j, xp\rangle$ *in* M *with* $w = w'$.

To simplify the approach we use the convention that for equation e_i at most one a-entry is contained in $\mathfrak{P}$ and that $\mathfrak{P}$ contains no p-entry with i as second component if an a-entry $\langle w, i, *, *\rangle$ exists. Like for p-entries, we can exactly reconstruct all overlap entries that are represented in an a-entry:

Lemma 7.4 *Provided the collected history of* $\mathfrak{A}$ *is available, the a-entry is of the form* $\langle w, i, *, *\rangle$*, and the current time is at least* i*, we can implement function* decompress *such that* $\mathsf{decompress}(\langle w, i, *, *\rangle) = M$ *iff* $\langle w, i, *, *\rangle$ *represents* M*.*

Proof From $\mathfrak{A}_i$ we can simply compute $C_i = \mathsf{CP}^{\succ}(e_i, \mathfrak{A}_i)$. Then we proceed as in Lemma 7.2. □

Similar to the compression into p-entries, we say that replacing the set of overlap entries $M = \{\langle w_1, i, j_1, xp_1\rangle, \ldots, \langle w_k, i, j_k, xp_k\rangle\}$ into the a-entry $\langle w, i, *, *\rangle$ is a *valid compression* iff the a-entry represents exactly M. That is $w = \min\{w_1, \ldots, w_k\}$ and $\mathfrak{P} - M$ contains neither an overlap entry $\langle w', i, j', xp'\rangle$ with $w' \ge w$ nor a p-entry of the form $\langle w', i, j', *\rangle$, nor an a-entry of the form $\langle w', i, *, *\rangle$. Hence, with valid compressions into a-entries, the invariant that each critical pair is represented exactly once is preserved. Again, the deleteMin operation has to be slightly modified to decompress a-entries when needed. We then get the desired result, which is proved analogously to Lemma 7.3.

Lemma 7.5 *Provided the collected history of* $\mathfrak{A}$ *is available, an implementation of* $\mathfrak{P}$ *using i-entries, p-entries, a-entries, and only valid compressions shows the same I/O-behavior as the reference implementation.* □

In our implementation we also need an incremental form of compression: We want to adjoin efficiently the overlap entry with the maximal weight to the corresponding a-entry. Let $\langle w, i, j, xp\rangle$ be the overlap entry and $\langle w', i, *, *\rangle$ be the corresponding a-entry. Then we can first replace the a-entry by $\mathsf{decompress}(\langle w', i, *, *\rangle)$ and then perform a

valid compression introducing the a-entry $\langle w, i, *, * \rangle$. As the ordering on weights is total and $\langle w, i, j, xp \rangle$ is the maximal overlap entry, the valid compression eliminates exactly $\langle w, i, j, xp \rangle$ and the result of decompress($\langle w', i, *, * \rangle$). Hence, we can achieve the same result simply by deleting $\langle w, i, j, xp \rangle$ and updating $\langle w', i, *, * \rangle$ to $\langle w, i, *, * \rangle$, which is of course more efficient.

Strategies for compression and reconstruction

In a third step, we will now sketch a realization of $\mathfrak{P}$ with space requirements that grow linearly with abstract time. For reasons of simplicity we use only i-entries and a-entries. The general idea is to use two main buffers B and B' and a temporary buffer T.[3] All buffers are priority queues implemented as array-based heaps. Buffer B serves as a *cache*. It has constant size and is used to store promising critical pairs in overlap entries. Buffer B' contains the rest of $\mathfrak{P}$. If equation e_i is interreduced, it contains the corresponding interreduction entry. If equation e_i is still active, B' may contain an a-entry that represents critical pairs with large weights belonging to e_i. Of course, it may happen that all critical pairs belonging to e_i are selected while e_i is still in $\mathfrak{A}$. Then B' contains no entry corresponding to e_i. In any case, for each activated fact B' contains at most one entry. Therefore, the size of B' grows linearly with abstract time.

After having set the main design decision, we consider the strategy for using the cache buffer B. This reduces to two questions: With which entries do we fill the buffer B? In case B is full, with what strategy do we replace entries of B? We first concentrate on the latter question. If B gets full, we first sort the entries of B according to their weights. Then B still fulfills the heap condition. Next, we delete any element in the "heavier" half of B in decreasing order and adjoin them to the corresponding a-entries in B'.

To fill buffer B, it is tempting to use any overlap entry available. However, this leads to many replacements operations, which makes this approach too costly. We therefore introduce an additional variable w_B. Initially, the value of w_B is set to ∞. After a reorganization of B, we set w_B to the maximal weight that B contains. We then insert overlap entries only if their weight is smaller than w_B. The overlap entries with a weight exceeding w_B are combined into one a-entry which is inserted into B'. To prevent B from running empty, we increase w_B to ∞ in case less than a quarter of B is occupied. Hence, w_B works as a damper for the fill level of B.

An insert operation works as follows. First, we consider all equations that are interreduced. Any a-entry belonging to some interreduced equation is deleted from B'. (This establishes a pro-active variant of the orphan-predicate). Then for each nonredundant interreduced equation we insert an interreduction entry into B'. After that, we consider the newly generated and already normalized critical pairs for which we store the corresponding overlap entries in the temporary buffer T. As long as the minimal entry of T has a weight that is smaller than w_B we insert this entry into buffer B. In case B gets full, we perform a reorganization of B according to the cache replacement strategy. If the minimal element of T has a weight that exceeds w_B, we compress the remaining entries of T into one a-entry and insert it into B'.

[3] We could do without T, but this would complicate the presentation.

If a deleteMin operation is performed then we compare the minimal entry of B with the minimal entry of B'. If the latter is smaller and consists of an a-entry we delete it from B' and perform a decompression operation. All i-entries represented by the a-entry a stored in T. We then proceed as during the insertion of critical pairs. Now the minimal entry is an individual entry. We take it out of the corresponding buffer and use it to reconstruct the equation it represents.

The whole approach trades time for space, but note that in general only the light-weighted entries ever need to be decompressed again.

THEOREM 7.1 *Provided the collected history of* $\mathfrak{A}$ *is available, the representation of* $\mathfrak{P}$ *based on* B *and* B' *shows the same I/O-behavior as the reference implementation and has space requirements that grow linearly with abstract time.*

PROOF The representation uses i-entries and a-entries, and all compressions performed are valid. Hence, by Lemma 7.5 the implementation is correct, it shows the same I/O-behavior as the reference implementation. The space requirements are clear, both buffers contain entries of fixed size, B has constant size, and B' grows linearly with abstract time. □

We can make better use of the space provided by buffer B if we use a min-max heap [ASSS86] instead of an ordinary heap.[4] Then in addition to the minimal element of B also the maximal element of B is quickly available. Hence, w_B becomes superfluous. When transferring elements form T to B and B is full we simply compare the weight of the minimal element of T with the maximal element of B. If the former is larger then we compress the remaining elements of T into one a-entry. Otherwise, we delete the maximal element of B, adjoin it to its corresponding a-entry, and insert the minimal element of T into B. Note that the invariant of representing each equation exactly once is still established. Therefore, Theorem 7.1 holds also for this refined approach. As this refined approach better utilizes the space in B we can reduce the size of B.

Supporting multi-dimensional heuristics

As previously discussed, $\mathfrak{P}$ should support multi-dimensional heuristics which are based on several weight functions $\varphi_1, \ldots, \varphi_k$. In a first attempt, we tried to adopt k-d-heaps [DW93] to our compression scheme using buffers B and B'. However, this turned out to be very complicated. The main problem is that the weight functions may assess an equation e very differently. One may consider e as "promising", some other quite the opposite. (This is exactly the whole point of using multi-dimensional heuristics). So the question arises: What are the "promising" equations to put into buffer B which serves as cache? Especially replacement strategies in case B gets full are difficult to design. As a result, the code became more and more involved and contained many special cases for obscure boundary conditions.

Hence we abandoned this branch of development and implemented the following approach which is noticeably simpler. For each weight function we use the two buffer

[4] The k-d-heaps of [DW93] provide a more general approach and can be used as a replacement.

approach based on B and B'. Hence, we have $2k$ buffers, B_1, ..., B_k of constant size, and B'_1, ..., B'_k that grow linearly. Of course, this needs k times the space, but usually only few weight functions are combined. Because we use several priority queues we have to ensure that no passive fact is delivered multiple times. Recall that each of the weight functions induces a total ordering. For each active fact e_i we keep a vector $\langle w_1, \ldots, w_k \rangle$ which for each dimension stores the weight of the maximal fact belonging to e_i that has been already selected according to that dimension. When we select an i-entry from one heuristics we check before returning it whether it has not alredy been selected by some other heuristics. Otherwise, we discard it and check the next i-entry. When finally an i-entry is returned we updated the corresponding weight vector.

Many multi-dimensional heuristics use φ_{fifo} as one of their weight functions. Because of the special nature of φ_{fifo}, there is no need to keep two buffers for this dimension. Instead, three counters for i, j, and xp suffice. These are increased similarly as for an odometer: For given i and j we increase xp until all extended positions are considered for this pair. Then we increase j by one and reset xp. When j surpasses i we increase i by one and reset j. Furthermore, it is unnecessary to keep a weight for φ_{fifo} in the weight vectors. Instead, we compare the components of the i-entry with the corresponding counters.

7.2.2 The set of active facts $\mathfrak{A}$ and its collected history

To provide the collected history of $\mathfrak{A}$ we have to solve essentially two problems. The first is to provide the activated equation e_i in unmodified form. The second is to efficiently provide the normal form functions Norm_i at time $j \geq i$. The first problem arises only because for efficiency reasons function $\mathsf{Interred}$ performs the interreduction of the right-hand sides of rules destructively. We therefore change function $\mathsf{Interred}$ to treat the interreduction of the right-hand sides analogously to the interreduction of left-hand sides. It turns out that this decision leads to further simplifications in the prover (see e. g. Section 7.4). The small investment in time leads to a remarkable win in architectural clarity.

It remains the problem to efficiently provide the normal form function Norm_i at time $j \geq i$. Usually the normal form of a term t is computed by traversing its subterms in some fixed order according to some *reduction strategy*. We stop at the first subterm that is an instance of the left-hand side of some active fact. If such a subterm is not found, t is irreducible; otherwise the subterm is replaced with the instance of the right-hand side, and the traversal starts all over again.

For such normal form functions, remembering Norm_i reduces to remembering the generalization retrieval with respect to $\mathfrak{A}_i$. In today's provers, this retrieval is based on some kind of indexing technique (see [Gra96] and [RSV01] for overviews). It is not necessarily unique, because more than one left-hand side may fit. For performance reasons *first-fit retrieval* is commonplace. That is, among the matching terms in the index, the first one found is selected. We therefore need to remember the corresponding generalization functions Match_i.

In the Waldmeister system, *perfect discrimination trees* are employed to index the active facts. That is a trie-like structure where terms are interpreted as sequences

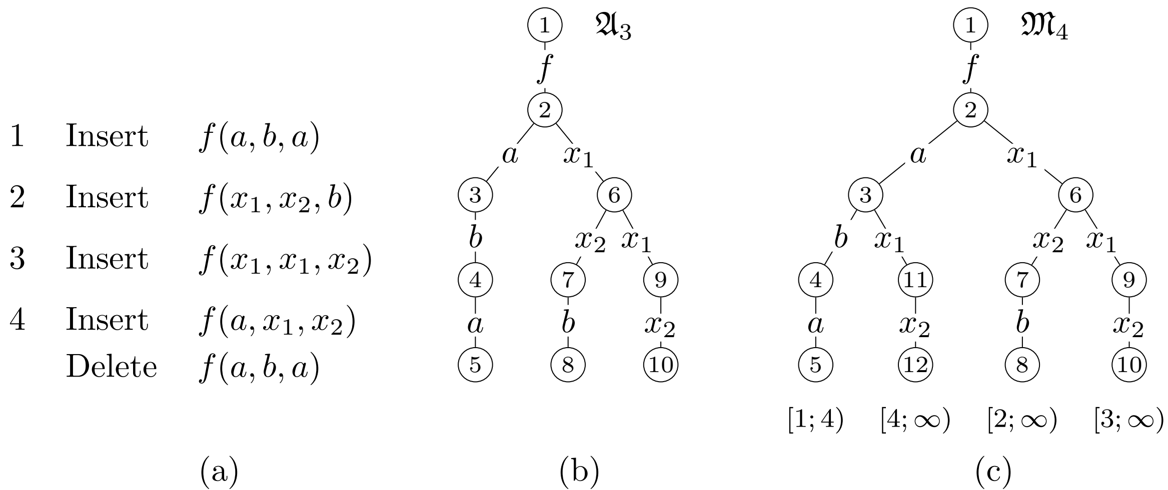


Figure 7.3: (a) Incremental construction of $\mathfrak{A}_1, \ldots, \mathfrak{A}_4$. (b) Index for $\mathfrak{A}_3$. (c) Index for $\mathfrak{M}_4$.

of symbols in left-to-right traversal, and every edge carries a symbol as a label. The construction principle is that, for every subtree, its term entries all have the same prefix, namely the sequence of labels on the path from the root to that very subtree. To increase the amount of this sharing of prefixes, variables of indexed terms are *normalized*, i. e., renamed to x_1, x_2, ... in order of their left-to-right fresh appearance.

The retrieval algorithm then works in a backtracking fashion on a triple of data (see [Hil00]): a pointer into the query term which is traversed in left-to-right order; a reference to the subtree to be searched through; and a partial substitution constructed so far. For every triple, there may be up to three possibilities to proceed: (i) The subterm starts with a function symbol, and the subtree has a descendant via an edge labeled with that very symbol. (ii) There is a descendant corresponding to a yet unbound variable. (iii) The same but with a variable already bound, such that its substitute is a prefix of the current query part. We stipulate that alternative possibilities are tested in that arrangement, i. e., first function symbols, and then variables by decreasing index. The algorithm succeeds in case it reaches a leaf node.

Example 7.1 *Consider the search for generalizations of $f(a,a,b)$ in the index for $\mathfrak{A}_3$ depicted in Figure 7.3 (b). This index contains the entries $f(a,b,a)$, $f(x_1,x_2,b)$, and $f(x_1,x_1,x_2)$. The retrieval starts with the empty substitution in node (1). There is a descendant via the top function symbol f; so (2) is reached with the query rest $a\,a\,b$. This node has two successors that might match: one via the constant symbol a, and another one via the variable x_1. According to the traversal strategy, (3) is visited first, where the query rest is $a\,b$. There is no suitable successor, so backtracking is invoked to (2) where the descent to (6) via the edge labeled x_1 is done, establishing a binding $\{x_1 \mapsto a\}$. The query rest is $a\,b$, and the node offers two possibilities to continue, namely with the free variable x_2 and the already bound one x_1. By convention x_2 is checked first, with binding $\{x_2 \mapsto a\}$, and thereby reaching (7) with query rest b. This is followed by the descent to (8) whereby a leaf node is reached, implying that the entry $f(x_1,x_2,b)$ matches $f(a,a,b)$ with the substitution $\{x_1 \mapsto a, x_2 \mapsto b\}$. Note*

that also the leaf node ⑩ corresponds to a successful match, namely $f(x_1, x_1, x_2)$ with $\{x_1 \mapsto a, x_2 \mapsto b\}$, but it is not the first one found.

The tree traversal induces an ordering on prefix symbol sequences in the index. This can be formalized with an appropriate total precedence $\succ$ on the function symbols $\mathcal{F}$, extended to $\mathcal{F} \cup \mathcal{V}$ via $f \succ x_j$ and $x_i \succ x_j$ iff $i >_{\mathbb{N}} j$. Then the lexicographic extension $\succ^*$ is a total ordering on symbol sequences, and also on terms considered as sequences of symbols. We then can formulate the following invariants for the retrieval process reaching a tree node:

1. The sequence of query term symbols considered so far equals the sequence of labels from the root to the current node under the substitution at hand; and hence every term indexed by the current subtree also has that prefix.

2. Consider a node which has a label sequence greater with respect to $\succ^*$ than the current one. Then the corresponding subtree contains a match only if it contains the current node.

For Example 7.1, when e. g. visiting node ⑥, this means first that the entries $f(x_1, x_2, b)$ and $f(x_1, x_1, x_2)$ under the substitution $\{x_1 \mapsto a\}$ start with $f\,a$, and second that the nodes ③, ④ and ⑤ do not lead to successful matches, but ① and ② might. These invariants guarantee not only that generalization retrieval is sound and complete, but also that the match found, if at all, is the greatest one with respect to $\succ^*$. That is, the algorithm without any overhead returns a uniquely determined match. We will exploit this property for efficiently remembering Match_i.

Corresponding to the state sequence of the active facts $\mathfrak{A} = \mathfrak{A}_1, \mathfrak{A}_2, \ldots$, we construct a *memory sequence* $\mathfrak{M} = \mathfrak{M}_1, \mathfrak{M}_2, \ldots$. This contains all the active facts that have ever been active so far, along with an additional activation and deactivation timestamp for each element. The key point now will be that a single additional discrimination tree is sufficient to represent $\mathfrak{M}$.

An insertion into the index for $\mathfrak{A}$ is reflected by a parallel insertion into the index for $\mathfrak{M}$, with an appropriate activation timestamp attached to the element. A deletion from the index for $\mathfrak{A}$, however, is *not* mirrored by a deletion from the index for $\mathfrak{M}$, but just by setting a deactivation time for the element. We need to define a generalization function $\mathsf{Match}_{i,j}$ that, if called at a current time $j \geq i$, uses the index for $\mathfrak{M}_j$ to reconstruct the operation Match_i once performed on the index for $\mathfrak{A}_i$.

For doing so, we only have to impose an additional, simple condition in the standard operation for the retrieval of generalizations: When a leaf node is reached, this implies success only if the attached activation and deactivation timestamps comply with i.

Example 7.2 *Figure 7.3 shows a sequence of activation and deactivation steps that results in an index for $\mathfrak{M}_4$ to capture the sequence of active fact sets $\mathfrak{A}_1, \ldots, \mathfrak{A}_4$. In Example 7.1 we have shown how a generalization for $f(a, a, b)$ is found in the index for $\mathfrak{A}_3$. This can now be reconstructed with the index for $\mathfrak{M}_4$: The retrieval operation successively visits the nodes ① and ② just like in the index for $\mathfrak{A}_3$. In node ③, there now is an appropriate successor via x_1 that leads to ⑪ and the leaf node ⑫. That*

node holds the matching entry $f(a, x_1, x_2)$, but its timestamps indicate that it is not in $\mathfrak{A}_3$. Therefore, backtracking to ② is invoked, and thereafter the nodes ⑥, ⑦ and ⑧ are inspected. That leaf node holds an entry with appropriate timestamps, and is hence returned. In fact, it is the one previously found in the index for $\mathfrak{A}_3$. Note that leaf node ⑩ which also corresponds to a successful match is not visited because the search is stopped at node ⑧.

The reason for this can be seen in the refinement of the second invariant for the adapted retrieval operation: Any node with a label sequence greater than that of the current one leads to a *timely* match only if it leads to the current node. Therefore this retrieval returns the $\succ^*$-greatest timely match in $\mathfrak{M}_j$. By construction of the $\mathfrak{M}$ sequence, the set of all timely matches is equal to the set of all matches in $\mathfrak{A}_i$. The uniqueness of the greatest match now implies that both operations return the same; so we have $\mathsf{Match}_{i,j} = \mathsf{Match}_i$. Note that the timestamp constraints could also be attached not only to leaves, but also to inner nodes, as union or covering of the timestamps of the descendants. In our example backtracking then would have been invoked not in node ⑫, but already in ⑪.

PROPOSITION 7.1 *Let $\mathfrak{A} = \mathfrak{A}_1, \mathfrak{A}_2, \ldots$ denote the sequence of sets of active facts and let $\mathfrak{M} = \mathfrak{M}_1, \mathfrak{M}_2, \ldots$ be the corresponding memory sequence. The indexing structures are perfect discrimination trees. Consider the abstract times i and j with $j \geq i$. Then $\mathsf{Match}_{i,j}(\mathfrak{M}_j, s) = \mathsf{Match}_i(\mathfrak{A}_i, s)$ for all terms s.* □

As it turns out, it is not necessary to keep two indices for $\mathfrak{M}$ and for $\mathfrak{A}$ for efficiency reasons. The former provides the functionality of the latter, and the slow-down usually observed in practice is in the order of very few percent. Hence, we use only $\mathfrak{M}$ and have no second index for the current $\mathfrak{A}_i$.

Let us finally remark that here a crucial prerequisite for efficiently remembering the generalization functions Match_i has been that the sets of all matches are naturally ordered such that the retrieval automatically stops with the unique greatest one. It is unclear to us how the same result can be achieved with other indexing schemes, especially if the index for a term set is not unique.

7.2.3 The new role of $\mathfrak{P}$

All in all, the elaborate scheme of compression and decompression results in a proof procedure that considerably departs from the original DISCOUNT loop. It is therefore adequate to give it a new name. Following [RV03] in using a well-known system implementing this loop, calling it "WALDMEISTER loop" seems appropriate.

From a more abstract point of view, the differences between DISCOUNT loop and WALDMEISTER loop can be interpreted as follows: The generation the critical pairs is delegated from the main loop to $\mathfrak{P}$. It is then the task of the subsystem $\mathfrak{P}$ to enumerate all nonredundant critical pairs in a fair way. $\mathfrak{P}$ is seen as a black box here. It may perform the same normalizations as the main loop did, but these are no "real" inferences, they are for heuristic purposes only. Conceptually, the "real"

inferences occur merely at the point of selection; only "good" critical pairs ever enter the completion loop.

Considering the formulation used for the logical foundations (see Section 3.2) we have come to a full cycle. The inference system $\mathcal{G}$ does *not* distinguish between active and passive facts, it considers only oriented rules and unoriented equations. All rules in R and all equations in E may participate in generating inferences, for simplification we may additionally use the elements of G. There are no passive facts in $\mathcal{G}$. If the role of $\mathfrak{P}$ is to fairly enumerate nonredundant critical pairs, it works as some kind of oracle, prophesying the parameters of the next DEDUCE-inference. This also explains why it is not sensible to enrich the initial normalization of critical pairs by more elaborate redundancy criteria (see Chapter 6). The use of these costly criteria becomes feasible only if we restrict their use to the promising critical pairs which is exactly what the WALDMEISTER loop does.

This view on the role of $\mathfrak{P}$ opens the avenue for further developments: Either one can replace the initial normalizations by some approximations. This influences the heuristics only. Or, one can completely depart from the given-clause algorithm. Using p-entries we can simulate, for instance, the *pair algorithm* [McC97a]. In this alternative of the given-clause algorithm a pair of activated equations e_i and e_j is selected via some heuristics ψ and then all critical pairs between them are generated. After simplification, all the nonredundant ones are activated and the next pair is determined. Reformulated in our framework, $\mathfrak{P}$ essentially consists of p-entries of the form $\langle \psi(e_i, e_j), i, j, * \rangle$. The weight function considers all i-entries as smaller than any p-entry. This ensures that after decompressing a p-entry, first all the i-entries are handled before the next p-entry is examined. If for an individual entry the values of ψ on the parents and φ on the equation are combined into one value, we get hybrid versions. We only sketch these ideas, we have not implemented them. The exploration of this design space is left for future research.

7.3 An experimental evaluation of the Waldmeister loop

Several important questions remain to be explored. Before an implementation we may ask: What is the expected trade-off between space savings and recomputation effort in practice? What are typical values for the average sizes of p-sets and a-sets? Is it worthwhile to modify core parts of existing systems for the integration of the new technique? After an implementation we may ask: What is the actual trade-off between space savings and recomputation effort in practice? Can the memory requirements be reduced without affecting running time too much? Does the WALDMEISTER loop improve the potential of the prover, i. e., does WALDMEISTER find proofs with the new approach it could not find before?

The first three questions can be answered without a full implementation of the method. To do so we retrofit WALDMEISTER to write run-time traces logging the equations inserted into or deleted from $\mathfrak{P}$. Then we can gather the desired data about the sizes of p-sets and a-sets from these traces. Furthermore, with their help we can *simulate* different compression schemes and can thus estimate the benefits of an im-

Table 7.1: Characteristic data of representative examples

example	activated facts	critical pairs			avg. size of p-sets	avg. size of a-sets
		generated	inserted	deleted		
GRP164-1	1 089	1 460 000	963 000	80 900	1.40	486
X6	1 408	2 760 000	2 650 000	47 200	3.64	1 960
RNG027-5	1 089	3 270 000	3 170 000	310 000	6.46	3 000
LAT038-1	2 753	3 760 000	1 630 000	179 000	1.68	1 370
ROB006-1	5 588	10 300 000	10 100 000	82 800	2.45	1 850
ROB001-1*	6 220	42 400 000	42 400 000	37 800	8.07	6 820

* The proof attempt of ROB001-1 was aborted because of memory limitations.

plementation. This is an important methodological step before undertaking a proper implementation which takes considerably more time than writing the simulation code. As the simulations were very promising, we can also answer the second group of questions and present "real" data that was measured using the full implementation of the new approach.

Writing and analyzing $\mathfrak{P}$-traces

We modify WALDMEISTER to write the following information into a file: for each generated critical pair the contributing parents, a unique identifier, and whether it can be shown redundant or whether it is inserted into $\mathfrak{P}$. For each element selected from $\mathfrak{P}$ the same identifier is written again. Each deletion of an element of $\mathfrak{A}$ is noted as well to allow a proper handling of interreductions and orphans in the simulations.

The proof problems that we consider are mainly taken from the TPTP library [SS98]. They are selected as representatives of different domains to cover a wide range. The only exception is the specification X6 which states that each ring with the identity $x^6 = x$ is commutative (similar to the easier TPTP-problem RNG009-5 which states the same for $x^3 = x$). For all these problems we choose parameter settings such that WALDMEISTER finds a proof within several minutes. As a contrast, we add the problem ROB001-1 for which WALDMEISTER does not find a proof and exits because of a memory overflow (even with the overlap term representation, see Figure 7.2).

In Table 7.1 we summarize relevant data extracted from the traces. Despite the fact that all the chosen examples generate more than a million critical pairs, the average size of p-sets is rather small, with a rather homogeneous distribution. This indicates that the use of p-entries for compression purposes is rather limited. Their real potential lies in the emulation and variation of the pair algorithm. The distribution of the sizes of the a-sets is far more widespread. After a short amount of abstract time we can expect them to be at least in the range of several hundreds of equations. So their use opens up the real opportunity for the drastic reduction of the space requirements.

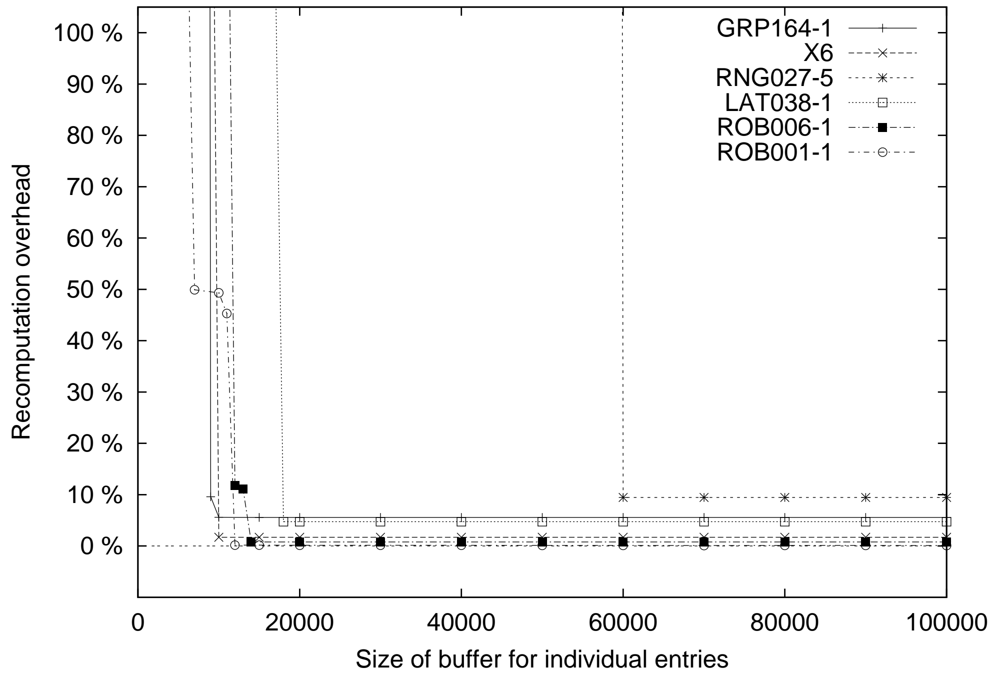


Figure 7.4: Simulated recomputation effort of the simple two buffer strategy as a function of the size of buffer B for the examples listed in Table 7.1.

Analyzing the strategies for compression and reconstruction

It is clear that the space requirements of both strategies described in Section 7.2.1 grow linearly with abstract time. However, it is *a priori* not clear what can be reasonable values for the size of B. When this value is chosen too small, we get a lot of decompression steps, which lead to considerable computation overheads. If we therefore have to choose a huge value for the size of B, the expected benefits of the whole method are endangered.

The complete implementation of the developed scheme requires a significant amount of developer time as core modules of a theorem prover are affected. Fortunately, it is possible to simulate the component $\mathfrak{P}$ in such a degree that we can estimate the trade-offs between space requirements and time requirements. The development of the simulator is a much smaller investment, it consists of about 300 lines of perl.

We use a very simple cost model and determine the amount of computation effort as the number of critical pairs which have to be computed and normalized. Thus, the costs for additional recomputations in case of a deleteMin operation or a decompression conversion are based on the number of affected equations. We choose this notion because in our experience the costs for each critical pair are quite evenly distributed. To get a useful measure across the different examples we set the computation effort of the simulated strategy in relation to that of the reference implementation.

Figure 7.4 shows the simulated recomputation effort as a function of the size of buffer B for the examples listed in Table 7.1. As can be seen, this straightforward strategy performs quite well. Even with a rather small value for the buffer size of say 20 000 all

Table 7.2: Comparison of resource utilization with and without set-based compression.

example	time [s]		size [MByte]		example	time [s]		size [MByte]	
	old	new	old	new		old	new	old	new
GRP164-1	67	68	71	13	LAT038-1	69	74	100	19
X6	141	137	213	13	ROB006-1	624	592	1012	21
RNG027-5	516	676	313	82	ROB001-1*	2940	3065	1115	23

* The proof attempt of ROB001-1 was aborted when $|\mathfrak{A}| = 7\,000$.

but one of the examples can be handled within a small computational overhead of less than 10 %. The exception is the proof run RNG027-5 which performs the most delete operations of all (see Table 7.1).

Comparing Figure 7.2 in Section 7.1 with the results of the simulation and taking into account the sizes of the different data structures, we can expect a huge reduction of the memory requirements. Instead of occupying 1 GByte of memory the new implementation should need less than 10 MByte, an improvement of two orders of magnitude for the whole system, and three orders of magnitude considering $\mathfrak{P}$ alone.

Experimental data from the new implementation

With the implementation of the new approach available, we can perform measurements to answer the remaining questions. The experiments were performed on machines with 1 GHz Pentium III processors and 4 GByte RAM.

Table 7.2 contains a comparison of the resource utilization of the old with the new implementation. For the size of buffer B we used a value of 40 000. As we can see, the memory requirements of Waldmeister are significantly reduced with the new approach. The additional work necessary to reconstruct the critical pairs that are selected from $\mathfrak{P}$ has only a moderate effect on the running times. In fact small speed-ups appear for some examples. We attribute this to the memory architecture of modern machines. With the new loop the processor caches are better used. Only for RNG027-5 a noticeable number of decompressions occur which is reflected in the running times. For this example, the chosen size of B is too small.

To analyze the dependency between the recomputation effort and the size of buffer B we ran Waldmeister with different settings for this value. Although the simulations do not model effects concerning the memory subsystem, they are pretty accurate. Compare Figure 7.5 which depicts the measured recomputation effort of the simple two buffer strategy with Figure 7.4 which contains simulated data. If the size of buffer B is large enough, the recomputation overhead is at most about 15 %. Depending on the example, it suddenly increases when the size of buffer B becomes too small. Figure 7.6 contains the data for the refined strategy. Apparently, the recomputation effort develops much more gracefully when the size of buffer B becomes smaller. This shows that the refined strategy uses the space provided in B in a much better way. Therefore, the refined strategy is now the standard approach taken in Waldmeister.

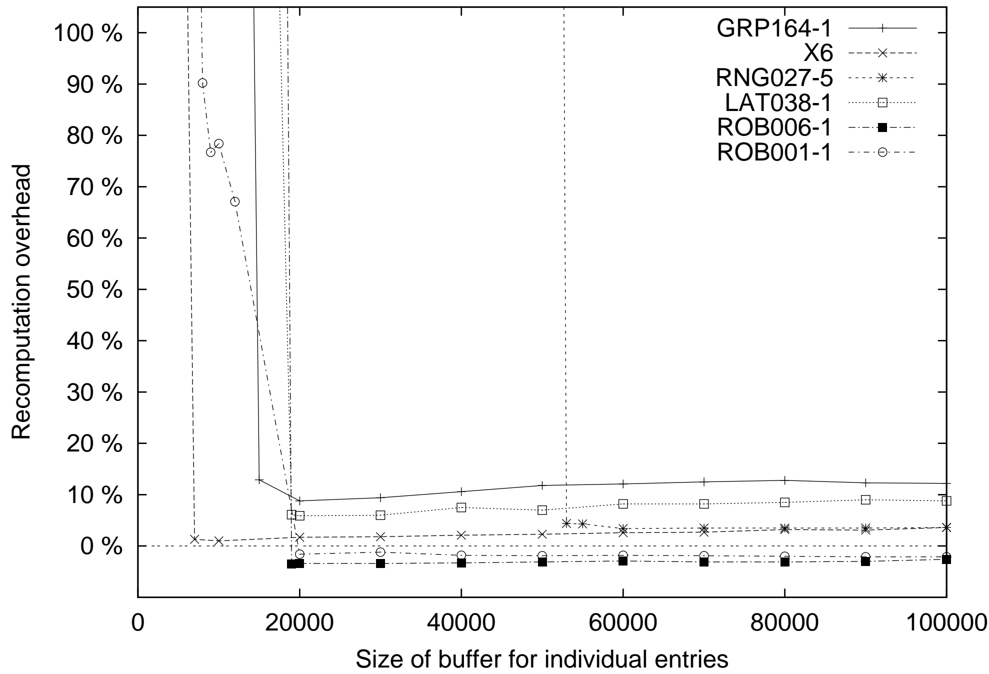


Figure 7.5: Measured recomputation effort of the simple two buffer strategy as a function of the size of buffer B for the examples listed in Table 7.1.

With the new architecture it is now possible to solve problems that were previously out of reach. So WALDMEISTER is now able to prove with simple standard heuristics that a ring with $x^5 = x$ is commutative (RNG036-7), or that the Robbins Axiom together with the second Winker Lemma entails Boolean (ROB007-1).[5] For these problems the size of $\mathfrak{A}$ exceeds 70 000, and more than 500 million equations are inserted into $\mathfrak{P}$. For such a large search state 200 MByte suffice, most of which are occupied by $\mathfrak{A}$. To our knowledge the only other system that can solve ROB007-1 is McCune's prover EQP [McC97b]. The EQP proof has been found with built-in AC-axioms and an incomplete strategy. Clauses exceeding a weight limit are discarded. It has been noted that too simplistic discarding heuristics are harmful to the success of the proof search. So some tuning knowledge concerning weight or clause limits is required, and it may be specific for problem domains or even individual problems. For the WALDMEISTER loop, however, such knowledge is not required.

However, we should note that the new approach is not a panacea. Of course, there are still interesting proof tasks for which WALDMEISTER fails to find a proof. An example is ROB001-1, the famous Robbins problem, which we used to illustrate the space requirements of the old implementation (see Figure 7.2). Figure 7.7 depicts the development of the memory requirements over abstract time. As expected (cf. Theorem 7.1), we can observe a clearly linear behavior. If we distinguish the four data structures that occupy most of the memory we see that $\mathfrak{P}$ needs now only 3.8 %. Most

[5] For running times of these and other challenging problems see Table 6.12 on p. 157.

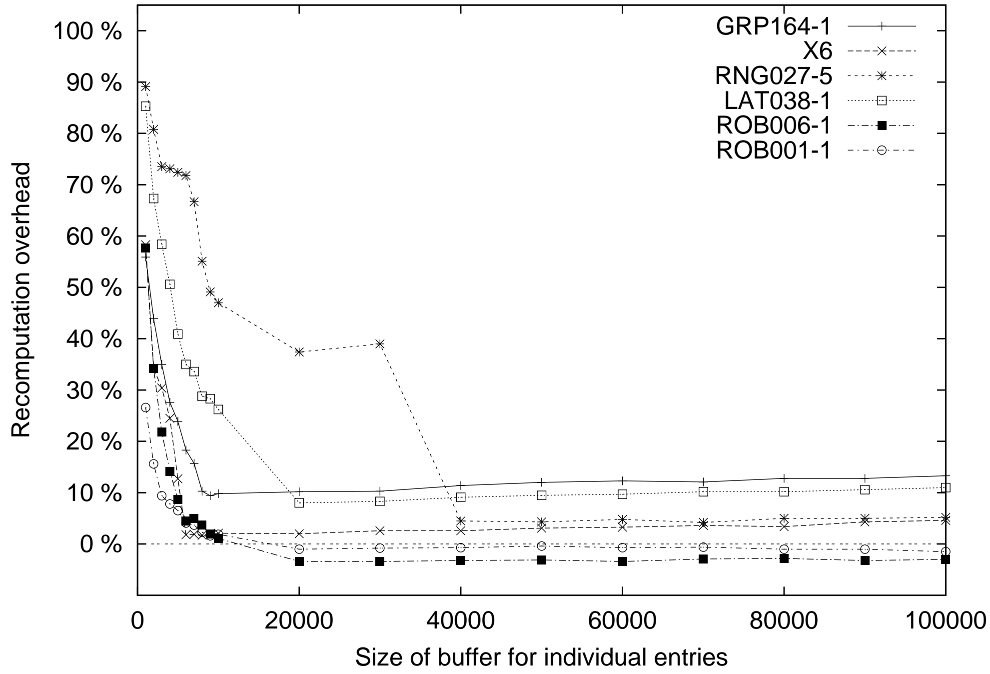


Figure 7.6: Measured recomputation effort of the refined two buffer strategy as a function of the size of buffer B for the examples listed in Table 7.1.

memory is now needed to represent $\mathfrak{A}$: 36 % for the terms of active facts, 50 % for the index of $\mathfrak{A}$, and 10 % for other data structures used by $\mathfrak{A}$. So now data structures are dominant that never attracted attention before. Therefore, they were never subject to a rigorous, space-concerned design.

When comparing Figure 7.7 with Figure 7.2, first note the different scales for both axes. At 5000 activated facts, the abstract time when even the most space-efficient old implementation exceeds 1 GByte, the new implementation needs only 7.5 MByte. The memory requirements are reduced by about two orders of magnitude. The new implementation exceeds 1 GByte after activating 824 000 equations. This means that the number of equations that can be activated before reaching this limit is increased by about two orders of magnitude. It is very interesting to relate abstract time with the *wall-clock time*. That is the (real) time that has passed since invoking the prover. In our example[6] the first situation is reached after 42 seconds, the second after 5.25 days. Analyzing the axis depicting wall-clock time (at the top of the diagram), we see that it roughly scales logarithmically with abstract time; the wall-clock time needed to make one step in abstract time increases considerably during the proof search. This can be attributed to the increasing size of $\mathrm{CP}(e_i, \mathfrak{A}_i)$. With the WALDMEISTER loop the combinatorial explosion concerns no longer space, but wall-clock time. This is illustrated by the statistics of this long-running proof attempt: Within 48 days, the system activated $2.1 \cdot 10^6$ equations, computed more than $1.8 \cdot 10^9$ critical pairs, performed $4.3 \cdot 10^9$ rewrite steps, and made $4.6 \cdot 10^{13}$ match attempts.

[6] The test run was performed on a machine equipped with 2.6 GHz Xeon P4.

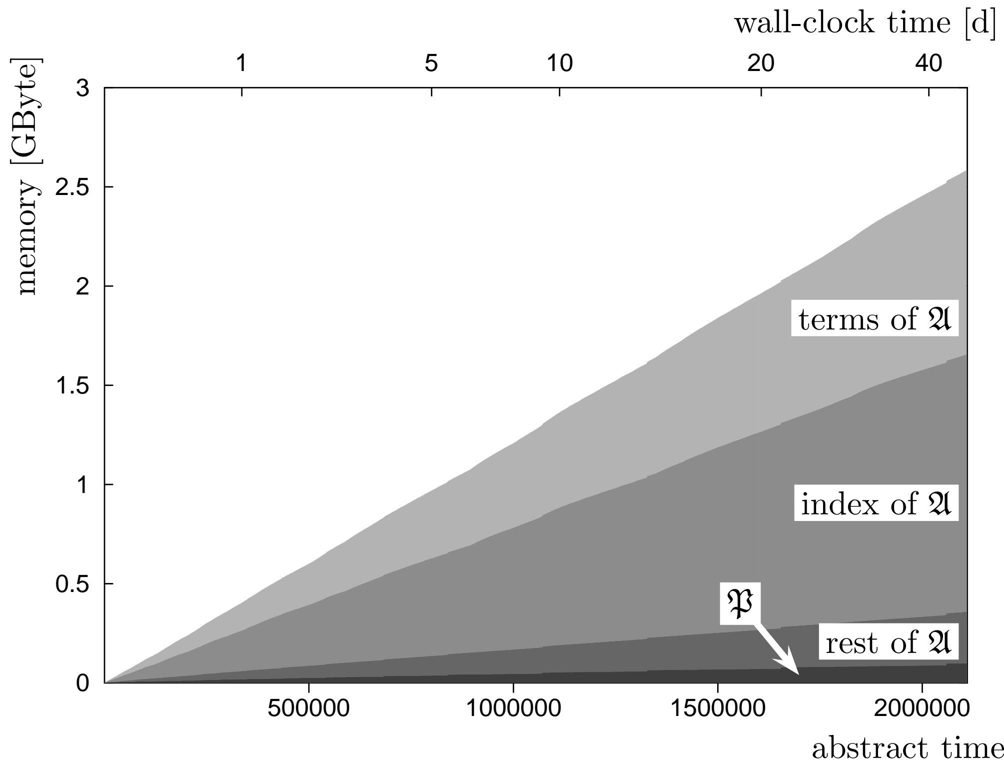


Figure 7.7: Memory requirements over time for ROB001-1: bottom axis denotes abstract time, top axis denotes wall-clock time in days

7.4 Further benefits

Besides the smaller memory footprint and some simplifications (e. g. for interreduction) there are some other benefits of the new loop design that are worth discussing. Not all of them are completely implemented yet.

Advantages of the compact search state representation

When we analyze the new representations of $\mathfrak{A}$ and $\mathfrak{P}$, we see that not only are they much more compact than the old ones, they are much simpler as well. For both, we can distinguish between the administrative data structures (the index of $\mathfrak{A}$ and the priority queues of $\mathfrak{P}$) and the actual contents. Given the contents, the administrative data structures are easy to construct. Hence, we can consider $\mathfrak{A}$ as a collection of rules and equations annotated with activation and deactivation timestamps; and the whole set $\mathfrak{P}$ consists only of tuples of numbers.

It is therefore much easier than with the old representation to develop routines that allow us to save proof states to disk and to restore them when necessary. With the old representation, this is simply not feasible because of the amount of data that would have to be written or read. Such a save/restore-facility is not only useful to protect long running proof searches against power failures, machine crashes, and kernel updates. They also give the further option to migrate to a different machine, or simply to stop the proof search with the option to resume it later.

Often, the completion of some (sub-)set of equations to a convergent rewrite system

is described as a form of compilation. However, for many subsets of interest there exists no finite convergent system. With the new representation it becomes feasible to pre-complete such subsets to some degree as all nonredundant critical pairs can be stored in $\mathfrak{P}$ in a compact way. This offers a new way to organize the prover. It can contain a library of some well-known and expensive to complete subsets that are pre-completed to some degree with regard to the most common orderings and weight functions. This is especially valuable if the prover is called repeatedly as subsystem from some other program, for which we know the typical structure of the input. As a concrete usage scenario, consider the recently discussed use of general purpose provers for decision procedures called from verification engines [ARR01, LM02].

Another recent development can be supported as well. Several researchers consider the combination of "small" verification engines [Sha02]. The main search is then performed by a DPLL-like procedure similar to a SAT-solver. This requires the verification engines to support backtracking [BGD03, SDK$^+$03]. The set of axioms does not increase monotonically. Instead, axioms are added and retracted later. With the compact representation of $\mathfrak{A}$ and $\mathfrak{P}$ the first time it becomes feasible to offer a backtracking facility for saturating provers. For each possible backtracking point, a fresh copy of $\mathfrak{P}$ is internally stored. Because the representation of $\mathfrak{P}$ is small, this is a cheap operation. When backtracking occurs, the copy is used to replace the then existing version of $\mathfrak{P}$. As the elements of $\mathfrak{A}$ are no longer modified destructively, backtracking to a certain point in history is even more straightforward for $\mathfrak{A}$. It suffices to delete elements that are too young and to restore the deactivation time stamps of elements that were interreduced.

A simple, yet powerful approach to strengthen the practical robustness of a prover is to *multiplex* the search. Instead of adhering to one search strategy, a time-slicing approach is chosen. When some given amount of time is exhausted the prover abandons the search so far and restarts with a different strategy. This is profitable, because many proof tasks can be quickly solved with one strategy, but are very hard for many others; and determining beforehand the right strategy is very difficult. For example, in the CADE system competition CASC, the prover GANDALF uses this approach [Tam97]. The new representation of $\mathfrak{A}$ and $\mathfrak{P}$ now allows us to refine the approach in two directions. First, when the time limit is reached it is not necessary to abandon the search completely. Instead, we can stop the search to resume it later. This leads to a refinement of the scheme. First small time slices are used. Later the stopped searches are resumed with an increasing amount of time. A second issue is that the different proof states are available in later stages. This enables the transfer of knowledge from one search to another; useful facts found in one search can help to improve the performance of an other search. As a result we get a sequential variant of the TEAMWORK approach which was originally developed for distributed search [AD93].

Proof objects for free

In theoretical expositions, theorem proving processes are described as semi-decision procedures. They are considered as partial functions returning a Boolean value to indicate whether the conjecture is a valid theorem. Figure 7.8 contains a pictorial representation of this view.

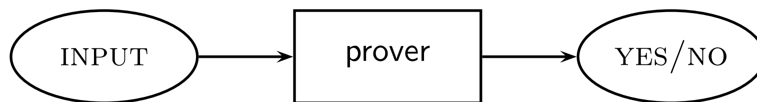


Figure 7.8: A theorem prover used as semi-decision procedure

However, the situation in practice is different. The prover does not run in isolation, but is called externally. We can distinguish three kinds of usage: Either the prover is called by some user who wants to know whether a conjecture is an actual theorem. Or the developer invokes the prover to test some data structure that was newly integrated or some new heuristics. Or an external program, such as a verification engine, calls the prover to solve some subproblem for it. In all three cases, a yes/no-answer is completely insufficient. The user wants to know why the conjecture actually is a theorem. The developer wants to know whether all steps performed were actually correct. And the external system wants to know which axioms were actually needed to prove the conjecture.

All three desires can be satisfied by returning a *proof object*. This is a machine-oriented, but human readable representation of the proof found by the system. Therefore, the external system can analyze the proof. The developer can check each step. And the user can be convinced that the conjecture is a theorem, usually with the use of a separate proof presentation program, see Figure 7.9. The proof object should be compact and contain only the steps really needed for the proof which usually amount to a tiny fraction of the steps performed during the search. Otherwise, the size becomes the limiting factor in processing the proof object. Furthermore, the proof object should be detailed enough to understand the proof and therefore contain for each step its justification. Some systems output only minimal proof objects omitting important details for which a reconstruction can be difficult.

High-performance provers use highly optimized algorithms and complex data structures. For efficiency reasons, they are usually coded in some rather low-level language such as C. For their development, only very limited resources are available. We can therefore safely assume that all systems currently available contain bugs; they are unsound. With today's techniques, it is simply impossible to formally verify such complex systems, at least, with the resources available. Proof objects pave an interesting way to solve this correctness problem. Compared to a full prover, a *proof checker* that verifies each step in a proof object is a small piece of software. As speed does not concern it can be developed in a high-level language. Hence, it is the ideal target for formal verification. If we combine a conventionally developed prover with a formally verified proof checker in the way depicted in Figure 7.10 we get a system that delivers only sound proofs. The Ivy system [MS00] follows this approach in combining the automated theorem prover Otter [McC05a] with some proof checker written and verified in ACL2 [KMM00]. Providing proof objects can be considered as an instance of *program result checking* [WB97], the importance of which is recognized in other areas as well. For example, it is used in the LEDA-library [MN98]; and Mehlhorn stresses the importance of reliable algorithmic software [Meh03].

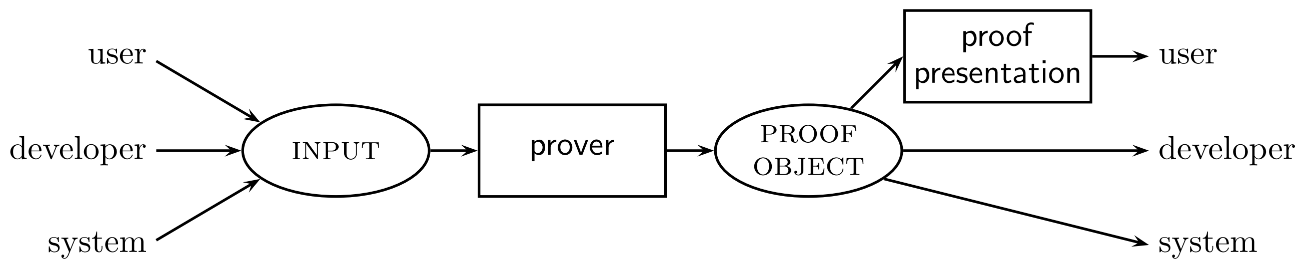


Figure 7.9: A theorem prover producing a proof object

All in all, the provision of compact and detailed proof objects, being considered some exotic feature several years ago, is now understood as an important feature of automated theorem provers. However, it is considered to be a task that is difficult to solve (cf. the invited talk by Voronkov at the ESFOR workshop in 2004). Simple approaches that log every proof step into some file, as for example done in the DISCOUNT system [DS94], are not feasible. They do not only tend to fill any disk space available, they also lead to such considerable slow-downs that high-performance proving becomes impossible. Both problems can be tackled with specialized techniques that keep the protocol internally in a highly-compressed form, extract the relevant steps after a proof has been found, and generate the proof object by reconstructing the relevant steps [Buc97]. This has the advantage to lead only to moderate slow-downs (about 5–10 %), but the data structure for the internal protocol can be the limiting factor for demanding proof tasks.

As it turns out, the new representation eliminates the need for internal protocols. Indeed, the WALDMEISTER loop offers proof objects for free. Recall that for $\mathfrak{A}$ we have the complete history available, especially, all elements in their original form (they are not modified destructively) and for any time i the corresponding normal form function Norm_i. When a proof has been found, the prover sets an internal flag such that the routines implementing the inferences write each step into the protocol and mark each involved fact as contributing for the proof. Then the rewrite proof of the conjecture is performed again. After that the proof object contains the steps that directly affect the conjecture; and all elements of $\mathfrak{A}$ that are involved are marked as contributing. Now, we traverse all elements of $\mathfrak{A}$ according their age, from the youngest (activated at late times) to the oldest (activated at early times). For each element marked as contributing, we reconstruct the original critical pair and re-perform the simplification steps once done before the heuristic assessment and before activating the equation. During this reconstruction, each step is added to the proof object and additional elements of $\mathfrak{A}$ may be marked as contributing. As they are all older than the current element, the loop will consider them later on. Hence, the necessary fixpoint computation is implicitly performed in this single loop.

Compared to our former approaches for generating proof objects, the new method is highly superior: (i) It does not need additional data structures. The provision of proof object does not lead to enhanced memory requirements anymore. (ii) The computational overhead is very small. Even for the largest proof objects we encountered the time for the proof object construction is a fraction of a second. (iii) The necessary

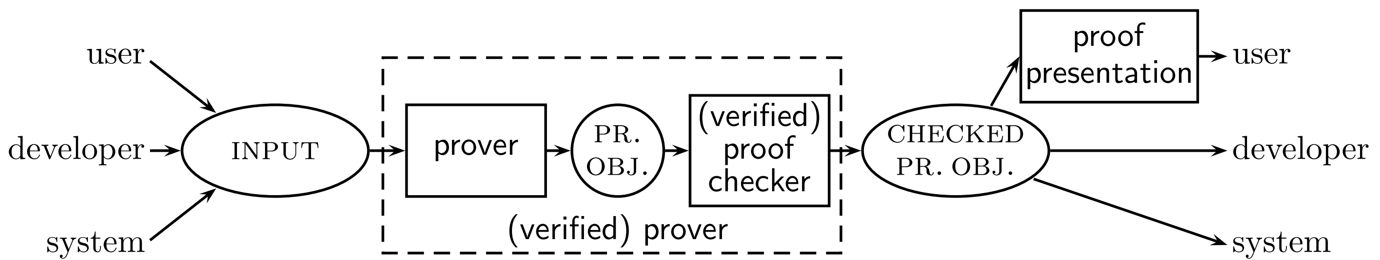


Figure 7.10: Verifying the proof checker leads to verified theorem prover.

code is quite short and simple. Straightforward checks in the reproduction code can already find bugs caused by violated invariants and corrupted data structures. As a most welcomed side-effect, we have an additional win in architectural clarity. Proof search and proof object construction are completely separated.

A parallel version of Waldmeister

During the last 50 years, sequential processor speed has developed enormously. The speed has doubled roughly every 18 months (see [HP03]). However, in the last years this development has slowed down. The main reason is that memory latencies have only slowly improved in comparison and that it becomes increasingly more difficult to hide them with caches. Hardware manufacturers therefore nowadays focus more on improvements in parallel speed. One recent trend is to allow the quasi-simultaneous execution of several threads on one processor (e. g. Intel's Hyperthreading technology). As soon as a thread stalls because of a memory access another thread is activated which leads to a better cpu utilization. A second trend is to simply put several processor cores on one chip. For example, recent IBM Power processors have two cores per chip. Intel and AMD have announced similar products, so even standard desktop PCs will be multi-processor machines in the near future. These developments amplify the effect of networked workstations and cheap 2- or 4-way servers: the average user has more and more parallel processing facilities at his or her fingertips. Furthermore, we can expect that, at least for server processors, multi-threaded performance will become the primary design goal, even at the expenses of single-threaded performance. As a consequence, for user acceptance it becomes more and more important that automated theorem provers can make efficient use of parallel hardware.

Very different approaches have been investigated for the use of multiple processors for automated theorem proving (see [Bon00] for an overview). It turns out that communication can be the major bottleneck: communication latencies may introduce delays in the computation, whereas communication bandwidth limits the amount of data transmitted. Variations in machine loads and in communication delays can further influence the proof search. Because the saturation process is unstable with regard to small disturbances our main design requirement is that the prover shall remain deterministic in order to ensure that proof runs are reproducible. This is in our experience of utmost importance for user acceptance.

From literature two main directions are of interest for us. The first path tries to speed up a sequential prover by *parallelizing* the most time-consuming subtasks (e.g. [BGKW98]). Because of the typically high communication demands this works best on shared memory machines. The second one is an instance of *multi-agent search*. Several instances of theorem provers cooperate by exchanging information (e.g. [DF99]). Some of these approaches are specifically designed to run on networked workstations or PCs.

We focus here on parallelizing WALDMEISTER because some kind of multi-agent search can be built on top later on. As the design focus of the system was on high-performance and not on easy parallelization this is somewhat challenging. But it turns out that the WALDMEISTER loop offers a new starting point.

WALDMEISTER spends most of its time in the computation, normalization, and heuristic assessment of critical pairs before they are inserted into $\mathfrak{P}$. Depending on the example, this amounts from about 80 % to over 90 % of the total running time, the longer the task runs the higher the percentage. The compact representation of critical pairs makes a good basis for parallelizing this part of the prover, as it offers a compact interchange format. So the communication bandwidth is not a limiting factor anymore. To achieve a deterministic behavior, we use an *asymmetric* form of parallelization: one process, the *master*, performs the saturation, whereas several *slaves* compute, normalize, and assess the individual critical pairs which constitute the smallest unit of parallelization. Thus, the parallelization happens above the inference level and basic routines such as unification or simplification are unaffected. The communication is completely based on a message-passing scheme: The master informs the slaves about the changes of $\mathfrak{A}$, whereas the slaves send back small, constant-size entries describing the nontrivial critical pairs and their weights.

In a first attempt we want to change the overall loop only minimally. When the master has activated an equation e_i, the n slaves process C_i. The work can be partitioned evenly among the slaves with some simple modulo scheme which gives a good load balance. The main problem with this approach is that it is very sensitive to communication delays. The master has to wait for all messages of the slaves before it can proceed, as it has to integrate all critical pairs belonging to equation e_i before it can select equation e_{i+1}. The whole saturation process is therefore stopped when one message is delayed. This limits the achievable speed-ups considerably in practice.

The key idea for hiding such delays is to *postpone* the integration of critical pairs by some fixed amount p, $p > 1$. That means that the critical pairs of equation i are integrated into $\mathfrak{P}$ just before equation e_{i+p} is selected. This changes of course the search behavior of the prover, but it changes it *deterministically* for a fixed value of p. This scheme has the further advantage that it can compensate variations in processing times of subsequent C_i-sets to some degree and even smooth out performance differences of the slaves. It is thus possible to simplify the approach, each C_i is handled only by one slave, the number of messages per set is cut down from n to two.

To evaluate our approach we implemented it in WALDMEISTER. As communication middleware we used PVM, version 3.4. We then made experiments on a cluster of machines equipped with 1 GHz Pentium III processors and 4 GByte of RAM connected via 100 MBit Fast Ethernet. The experiments show that for long-lasting runs a moderate value of p does not change the search process significantly. For example, for the

Table 7.3: Experiments with parallel WALDMEISTER

	ROB020-1				ROB007-1			
number of processors	1	3	5	7	1	3	5	7
number of slaves	0	2	4	6	0	2	4	6
cpu time of master [h]	8.4	0.9	0.9	0.9	43.6	4.4	4.5	4.4
average cpu time of slaves [h]	—	3.6	1.8	1.2	—	18.7	9.3	6.1
wall-clock time (wct) [h]	8.4	3.7	1.9	1.4	43.8	21.0	10.1	7.5
speed-up (w.r.t. wct)	1.0	2.3	4.4	5.9	1.0	2.1	4.3	5.8

proof task ROB020-1 the size of $\mathfrak{A}$ increases from 36 976 to 37 135 when $p = 30$. In effect, as can be seen from Table 7.3, the parallel WALDMEISTER gains nice, nearly linear speed-ups. The work is distributed quite evenly between the different slaves. Our approach does hide the communication latencies quite well, as the small differences between the maximal cpu time and the wall-clock time indicate.[7] With six or seven slaves, the master is not (yet) the bottleneck.[8]

In Figure 7.11 we have depicted the speed-up over abstract time for proof task ROB020-1 using different numbers of slaves. For each number, the curves first rise steadily to reach their limit at about 10 000. Then it stays roughly constant. This shows some limitation of our approach with a static number of slaves. During the initial phase, enough work suitable for parallelization is not yet available. The advantages of adding a slave become smaller with an increasing number of slaves, as the distances between the different curves indicate. Whereas two slaves utilize their machines to about 98 %, with seven slaves this drops down to 70 %. This indicates that then there is not sufficient work available for the slaves. We expect that this can be changed by increasing the value of p.

All in all, these are very respectable results for a high-performance, originally sequential prover running on standard desktop PCs that are interconnected by standard Ethernet. The WALDMEISTER loop is the key for enabling this parallelization scheme. While testing WALDMEISTER in parallel for some problems we found for the first time a proof – using overnight the idle cpus of our colleagues' PCs.

7.5 Related work and concluding remarks

A nice overview of the architectures of different provers developed in Argonne over the years is presented in [Lus92]. The issue of main-loop design is also discussed in [Wei01] and [Vor01] (cf. our discussion on p. 161). This chapter is based on work first presented in [HL02] and [GHLS03].

[7] The relationship between both numbers is better for ROB020-1 than for ROB007-1. This indicates that for the latter a greater value of p might lead to better speed-ups.

[8] Unfortunately, we had no more equivalent machines available, so we could not perform experiments with more slaves.

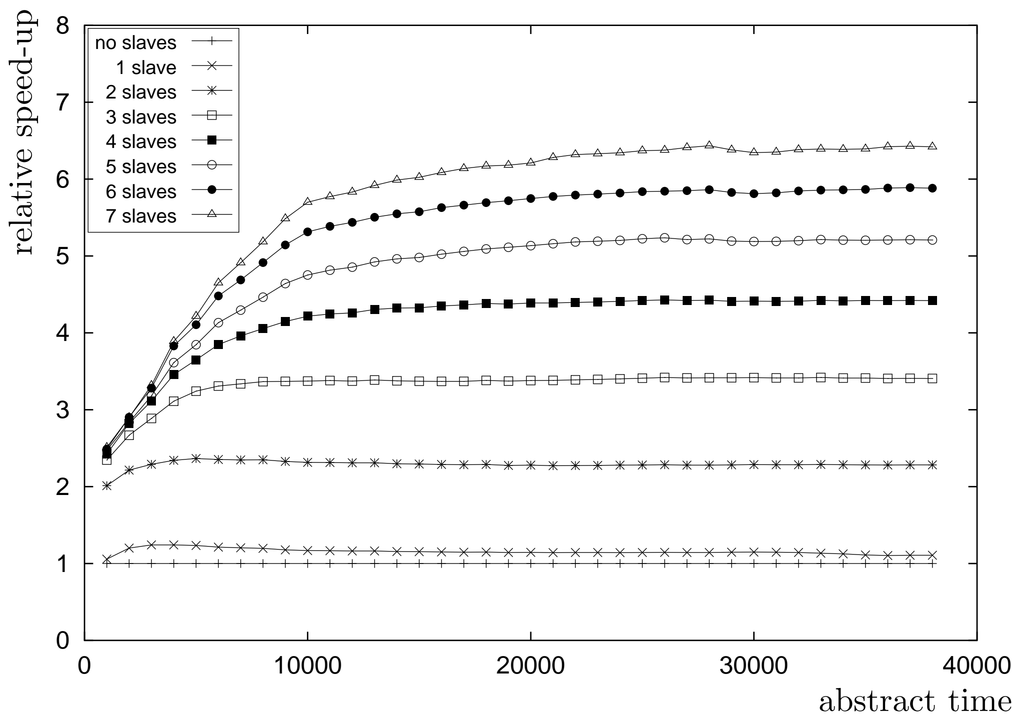


Figure 7.11: Development of speedups during the proof search for ROB020-1 with 0 to 7 slaves.

Interesting related work on the resource problem in theorem proving can be found in [RV03]. The setting there is to attack a proof problem *within some given time limit*. This is different to our approach which aims at long-lasting proof runs. A *limited resource strategy*, LRS for short, is developed that explicitly takes this limitation into account in order to analyze a larger part of the search space in the time given. To this end, the prover keeps track of the average time spent for processing a selected passive fact. This allows to estimate the number N of facts that still can be activated before the time limit is reached. Hence $|\mathfrak{P}| - N$ passive facts are not likely to be selected; and so the $|\mathfrak{P}| - N$ heaviest ones of them are discarded. LRS is a refinement of the weight-limit strategy of OTTER where clauses are discarded if their weight exceeds a given, fixed bound. For the VAMPIRE system the authors observe an increase in performance with LRS. The approach pays off for systems implementing the OTTER loop, because fewer passive facts have to be kept and indexed. As to the DISCOUNT loop, space savings are to be expected there, but no speed-ups, because passive facts are not indexed.

Of course, our whole approach using a set-based compression scheme is suitable for reasoning with full clauses as well. The only requirement is that any sequence of simplification steps and redundancy tests performed on a passive fact at generation time can *identically* be reproduced at selection time. For binary resolution, the entry format simply has to be adapted such that the position information encodes the literal resolved upon. The extension to superposition or hyperresolution is an easy exercise as well. The more complex the logical objects dealt with are, the larger the benefits of the approach, such as for example with clauses augmented with constraints.

Exemplified by our system, the WALDMEISTER loop architecture shows a way to significantly exapand the limits saturation-based provers have been afflicted with. It has a clear distinction between active facts and passive facts. As relatively few passive facts are activated, the loop allows us to apply rather costly redundancy criteria (see Chapter 6). As passive facts are truly passive and do not participate in any inferences, we can considerably reduce the space requirements of the prover by employing a set-based compression scheme. Less constricted by memory and accelerated by the parallelization the prover has solved several problems that were previously out of reach. It can now run for days or weeks without filling up the memory of the machines. The WALDMEISTER loop is essential for solving the challenging proof tasks depicted in Table 6.12 in Chapter 6 (see p. 157). Further benefits are the simplification of the architecture, the cheap provision of proof objects, and the support for backtracking. This opens the way for new applications, for example in the context of verification engines.

8 Conclusions

Starting from a level-based system model we presented various techniques that are important for the development of a high-performance theorem prover. The main contributions of this work can be summarized as follows.

Architecture. With the WALDMEISTER loop we have dramatically reduced the memory requirements of a saturating prover. We have proved that this new representation leaves the proof search invariant. It is based on a novel set-based compression scheme of the passive facts. The additional computational costs are very moderate. The clear distinction between active facts and passive facts allows us to apply rather costly redundancy criteria for the relatively few equations that get activated. Several further benefits are worth mentioning: Detailed and compact proof objects can be generated without further overhead. The compact representation of facts helps to avoid communication bottlenecks; the idea to postpone the integration of facts enables parallel work and hides communication latencies. Thus, the parallel variant of WALDMEISTER shows very respectable speed-ups. With the compact representation available, several other developments become possible, such as saving and restoring the state, multiplexing the search, and introducing a limited form of backtracking.

Algorithms. In a detailed step-by-step manner we developed efficient implementations of LPO and KBO. To our knowledge, we presented the first implementation of KBO with linear time requirements. The use of algebraic specification and program transformation techniques has led to well-structured and error-free programs. For deciding the satisfiability of LPO constraints and for detecting ground joinability we inspected nondeterministic algorithms from the literature [NR02, MN90, CNNR03]. The strategies and optimizations that we developed make these algorithms applicable in practice. We presented a novel test for the unsatisfiability of ordering constraints. This test is of polynomial time complexity. Despite its genericity it is pretty accurate and for KBO constraints the strongest test currently available.

Redundancy avoidance. Our refinement of the unfailing completion approach allows us to use redundant equations for simplification without the need to consider them for computing critical pairs. The extension and refinement of the proof ordering makes more proofs comparable which leads to more redundant equations and thus to stronger redundancy criteria. All the criteria are justified by a strengthened form of ground joinability. As we showed, ground joinability and related problems are undecidable for ordered rewriting. This result answers an open question posed in [CNNR03]. We showed that ground reducibility can be used instead of ground joinability when the necessary overlaps are added. For the AC-case we developed an elegant sufficient criterion based on ordering constraints which shows an excellent balance between de-

tection strength and computational cost. The experiments document that redundancy avoidance leads to a significant reduction in running time.

As a result of the combination of all the techniques developed in this thesis, the prover can now solve problems that were previously out of reach. This considerably enhances the potential of the prover and opens up the way for new applications.

Future research directions

The development of an efficient and practically useful theorem prover is a moving target. So current answers may not be definite answers. Advances in hardware systems, in the theoretical foundations, in search heuristics, and of course in the implementation itself lead to new algorithmic questions and to new bottlenecks that emerge in practice.[1] Also new calculi need new algorithmic solutions (e. g. the Model Evolution Calculus [BFT04]). Finding efficient solutions for basic operations will probably always be a topic in the development of automated theorem provers.

A recent important trend is the integration of decision procedures for certain theories (such as linear arithmetic). They pose new questions to the overall design of the provers. As the proper theoretical foundations are still not settled it is currently unclear how this influences the main prover architecture. Conversely, general purpose provers can be employed to serve as decision procedure for certain theories [ARR01, LM02]. These usage scenarios require more than the usual black-box integration. The calling systems need a very fine control of the proof process to profit from domain knowledge that cannot properly be encoded in the input specifications of the provers. This can be condensed to the question: How do we open the provers?

In our opinion, the most promising approach for advancing general automated theorem provers is the development of new redundancy criteria and more robust search heuristics. They probably have the most potential to improve the practical usefulness of existing systems. Considering the growth of parallel hardware resources, some kind of multi-agent search may be helpful here (cf. [DF99]). Of course, the parallelization of the provers themselves should be pursued as well.

Summing up, there is still much work to do. As automated equational theorem proving is a semi-decidable problem, there is an infinite need for further improvements and additional speed-ups.

[1] Consider for example the representation of the active facts, which are now responsible for more than 95 % of the memory requirements (see Figure 7.7 on p. 183). They were never optimized for size, because they needed less than 5 % of the memory with the old prover architecture.

Bibliography

[AD93] J. Avenhaus and J. Denzinger. Distributing equational theorem proving. In C. Kirchner, editor, *Proceedings of the 5th Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 62–76. Springer, 1993.

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *LNCS*, pages 397–402. Springer, 1995.

[AHL03] J. Avenhaus, T. Hillenbrand, and B. Löchner. On using ground joinable equations in equational theorem proving. *Journal of Symbolic Computation*, 36(1–2):217–233, 2003.

[AKSSW03] J. Avenhaus, U. Kühler, T. Schmidt-Samoa, and C.-P. Wirth. How to prove inductive theorems? QUODLIBET! In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, volume 2741 of *LNCS*, pages 328–333. Springer, 2003. See `http://www-avenhaus.informatik.uni-kl.de/quodlibet.html`.

[AL01] J. Avenhaus and B. Löchner. CCE: Testing ground joinability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning*, volume 2083 of *LNCS*, pages 658–662. Springer, 2001.

[ARR01] A. Armando, S. Ranise, and M. Rusinowitch. Uniform derivation of decision procedures by superposition. In L. Fribourg, editor, *Proceeding 15th Workshop Computer Science Logic*, volume 2142 of *LNCS*, pages 513–527. Springer, 2001.

[ASSS86] M. D. Atkinson, J.-R. Sack, B. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.

[Ave95] J. Avenhaus. *Reduktionssysteme*. Springer, 1995.

[Bac91] L. Bachmair. *Canonical Equational Proofs*. Birkhäuser, 1991.

[BB03] P. Blackburn and J. Bos. Computational semantics. *Theoria*, 18(1):27–45, 2003.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BD88] L. Bachmair and N. Dershowitz. Critical pair criteria for completion. *Journal of Symbolic Computation*, 6:1–18, 1988.

[BD94] L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *Journal of the ACM*, 41(2):236–276, 1994.

[BDH86] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357, 1986.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[BDP89] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In *Resolution of Equations in Algebraic Structures*, volume 2, Rewriting Techniques, pages 1–30. Academic Press, 1989.

[Bel57] R.E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BFT04] P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin: A Theorem Prover for the Model Evolution Calculus. In S. Schulz, G. Sutcliffe, and T. Tammet, editors, *Proceedings of the 1st Workshop on Empirically Successful First Order Reasoning (ESFOR '04)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[BG98] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[BGD03] S. Berezin, V. Ganesh, and D.L. Dill. An Online Proof-Producing Decision Procedure for Mixed-Integer Linear Arithmetic. In H. Garavel and J. Hatcliff, editors, *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2618 of *LNCS*, pages 521–536. Springer, 2003.

[BGKW98] R. Bündgen, M. Göbel, W. Küchlin, and A. Weber. Parallel term rewriting with PaReDuX. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume 9 (2) of *Applied Logic Series*, chapter 9, pages 231–260. Kluwer Academic Publishers, 1998.

[BGLS95] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[BH95] M.P. Bonacina and J. Hsiang. Towards a foundation of completion procedures as semidecision procedures. *Theoretical Computer Science*, 146(1/2):199–242, 1995.

[BH96] A. Buch and T. Hillenbrand. WALDMEISTER: Development of a High Performance Completion-Based Theorem Prover. SEKI-Report 96-01, Universität Kaiserslautern, 1996.

[Bir35] G. Birkhoff. On the structure of abstract algebras. In *Proceedings of the Cambridge Philosophical Society*, volume 31, pages 433–454, 1935.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1/2):203–216, 1987.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Bon00] M. P. Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):223–257, 2000.

[BRLP98] J. Van Baalen, P. Robinson, M. R. Lowry, and T. Pressburger. Explaining synthesized software. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, pages 240–248, 1998.

[Bro75] T. Brown. *A structured design-method for specialized proof procedures.* PhD thesis, California Institute of Technology, 1975.

[BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[BTV03] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.

[Buc97] A. Buch. Über Platzeffizienz beim Gleichheitsbeweisen – Analysen, Entwurf und Implementierung. Diplomarbeit, Universität Kaiserslautern, 1997.

[CJ97] H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. In *Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 26–34, 1997.

[CNNR98] H. Comon, P. Narendran, R. Nieuwenhuis, and M. Rusinowitch. Decision problems in ordered rewriting. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 276–286, 1998.

[CNNR03] H. Comon, P. Narendran, R. Nieuwenhuis, and M. Rusinowitch. Deciding the confluence of ordered term rewrite systems. *ACM Transactions on Computational Logic*, 4(1):33–55, 2003.

[Com90] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–412, 1990.

[CT94] H. Comon and R. Treinen. Ordering constraints on trees. In S. Tison, editor, *19th International Colloquium on Trees in Algebra and Programming*, volume 787 of *LNCS*, pages 1–14. Springer, 1994.

[CT97] H. Comon and R. Treinen. The first-order theory of lexicographic path orderings is undecidable. *Theoretical Computer Science*, 176:67–87, 1997.

[DF99] J. Denzinger and D. Fuchs. Cooperation of heterogeneous provers. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 10–15. Morgan Kaufmann, 1999.

[DFS04] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In D. A. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *LNCS*, pages 198–212. Springer, 2004.

[Dom92] E. Domenjoud. A technical note on AC-unification. The number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p \doteq_{AC} \beta y_1 + \cdots + \beta y_q$. *Journal of Automated Reasoning*, 8:39–44, 1992.

[DP01] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier, 2001.

[DS94] J. Denzinger and S. Schulz. Recording, Analyzing and Presenting Distributed Deduction Processes. In H. Hong, editor, *Proc. 1st PASCO, Hagenberg/Linz*, volume 5 of *Lecture Notes Series in Computing*, pages 114–123, Singapore, 1994. World Scientific Publishing.

[DW93] Y. Ding and M. A. Weiss. The k-d heap: An efficient multi-dimensional priority queue. In J-R. Sack, N. Santoro, and S. Whitesides, editors, *Proceedings of Third Workshop on Algorithms and Data Structures, WADS '93*, volume 709 of *LNCS*, pages 302–313. Springer, 1993.

[EPW03] U. Egly, R. Pichler, and S. Woltran. On deciding subsumption problems. Technical report, Technische Universität Wien, 2003.

[Eva51] T. Evans. On multiplicative systems defined by generators and relations. I. Normal form theorems. *Proceedings of the Cambridge Philosophical Society*, 47:637–649, 1951.

[FL96] C. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6:173–230, 1996.

[FMS02] R. Fleischer, B. Moret, and E. M. Schmidt, editors. *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, volume 2547 of *LNCS*. Springer, 2002.

[FW86] P.J. Fleming and J.J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[GCL92] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic, 1992.

[GHLS03] J.-M. Gaillourdet, T. Hillenbrand, B. Löchner, and H. Spies. The new Waldmeister loop at work. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction*, volume 2741 of *LNCS*, pages 317–321. Springer, 2003.

[GN01] H. Ganzinger and R. Nieuwenhuis. Constraints and theorem proving. In H. Comon, C. Marché, and R. Treinen, editors, *Constraints in Computational Logics: Theory and Applications*, volume 2002 of *LNCS*, pages 159–201. Springer, 2001.

[Gra96] P. Graf. *Term Indexing*, volume 1053 of *LNCS*. Springer, 1996.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19/20:583–628, 1994.

[Hil00] T. Hillenbrand. Schnelles Gleichheitsbeweisen: Vom Vervollständigungskalkül zum Waldmeister-System. Diplomarbeit, Universität Kaiserslautern, Fachbereich Informatik, 2000.

[HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. System description: Waldmeister – improvements in performance and ease of use. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *LNCS*, pages 232–236. Springer, 1999.

[HL02] T. Hillenbrand and B. Löchner. The next Waldmeister loop. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 486–500. Springer, 2002.

[HP03] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A quantitative Approach*. Morgan Kaufman, third edition, 2003.

[HR87] J. Hsiang and M. Rusinowitch. On word problems in equational theories. In *Proceedings 14th International Colloquium on Automata, Languages and Programming*, volume 267 of *LNCS*, pages 54–71. Springer, 1987.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):798–821, 1980.

[Hue81] G. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 32(1):11–21, 1981.

[Hur03] J. Hurd. Using inequalities as term ordering constraints. Technical Report 567, University of Cambridge Computer Laboratory, 2003.

[Jae97] A. Jaeger. Anwendungen von Nachfolgermengen in Termersetzungssystemen. Projektarbeit, Universität Kaiserslautern, Fachbereich Informatik, 1997.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[JK89] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989.

[JSA94] P. Johann and R. Socher-Ambrosius. Solving simplification ordering constraints. In J.-P. Jouannaud, editor, *First International Conference on Constraints in Computational Logics CCL*, volume 845 of *LNCS*, pages 352–367. Springer, 1994.

[Kap97] D. Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Proc. 8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 23–37. Springer, 1997.

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Algebra*, pages 263–297. Pergamon Press, 1970.

[KKR90] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990.

[KL80] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Departement of Computer Science, University of Illinois, Urbana, IL, 1980.

[KMM00] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. System's home page: `http://www.cs.utexas.edu/users/moore/acl2/`.

[KN92] D. Kapur and P. Narendran. Double-exponential complexity of computing a complete set of AC-unifiers. In *Proceedings of the Seventh Symposium on Logic in Computer Science*, pages 11–21, 1992.

[KNO90] D. Kapur, P. Narendran, and F. Otto. On ground-confluence of term rewriting systems. *Information and Computation*, 86:14–31, 1990.

[KNS85] D. Kapur, P. Narendran, and G. Sivakumar. A path ordering for proving termination of term rewriting systems. In H. Ehrig et al., editors, *Colloquium on Trees in Algebra and Programming*, volume 185 of *LNCS*, pages 173–187. Springer, 1985.

[KNZ87] D. Kapur, P. Narendran, and H. Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(395–415), 1987.

[KS95] D. Kapur and G. Sivakuma. Maximal extensions os simplification orderings. In P. S. Thiagarajan, editor, *15th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 225–239. Springer, 1995.

[Küc82] W. Küchlin. An implementation and investigation of the Knuth-Bendix completion procedure. Interner Bericht 17/82, Universität Karlsruhe, Fakultät für Informatik, 1982.

[KV00] K. Korovin and A. Voronkov. A decision procedure for the existential theory of term algebras with the Knuth-Bendix ordering. In *Proc. 15th Annual IEEE Symp. on Logic in Computer Science (LICS'00)*, pages 291–302, 2000.

[KV01] K. Korovin and A. Voronkov. Knuth-Bendix constraint solving is NP-complete. In *Proceedings of 28th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *LNCS*, pages 979–992. Springer, 2001.

[KV02] K. Korovin and A. Voronkov. The decidability of the first-order theory of the Knuth-Bendix order in the case of unary signatures. In *Proceedings of the 22th Conference on Foundations of Software Technology and Theoretical Computer Science, (FSTTCS'02)*, volume 2556 of *LNCS*, pages 230–240. Springer, 2002.

[Lan75] D.S. Lankford. Canonical inference. Technical Report ATP-32, Department of Mathematics and Computer Science, University of Texas, Austin, 1975.

[LB77] D.S. Lankford and A.M. Ballantyne. Decision procedures for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions. Technical Report ATP-39, University of Texas, Austin, 1977.

[LH02] B. Löchner and T. Hillenbrand. A phytography of WALDMEISTER. *AI Communications*, 15(2–3):127–133, 2002. See `http://www.waldmeister.org`.

[LM02] C. Lynch and B. Morawska. Automatic decidability. In *Proceeding of the 17th IEEE Symposium on Logic in Computer Science*, pages 7–16, 2002.

[Löc04a] B. Löchner. A redundancy criterion based on ground reducibility by ordered rewriting. In D. A. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *LNCS*, pages 45–59. Springer, 2004.

[Löc04b] B. Löchner. Things to know when implementing LPO. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the 1st Workshop on Empirically Successful First Order Reasoning (ESFOR '04)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[LS95] C. Lynch and W. Snyder. Redundancy criteria for constrained completion. *Theoretical Computer Science*, 142:141–177, 1995.

[LS01] B. Löchner and S. Schulz. An evaluation of shared rewriting. In H. de Nivelle and S. Schulz, editors, *Proceedings of the Second International Workshop on Implementation of Logics*, Technical Report MPI-I-2001-2-006, pages 33–48. Max-Planck-Institut für Informatik Saarbrücken, 2001.

[Lus92] E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Proceedings of the 3rd International Conference on Logic Programming and Automated Reasoning*, volume 624 of *LNCS*, pages 96–106. Springer, 1992.

[McC92] W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[McC97a] W. McCune. 33 basic test problems: a practical evaluation of some paramodulation strategies. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 5, pages 71–114. MIT Press, 1997.

[McC97b] W. McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McC05a] W. McCune. Otter: An Automated Deduction System. System's home page: `http://www-unix.mcs.anl.gov/AR/otter/`, 1988–2005.

[McC05b] W. McCune. EQP: Equational Prover. System's home page: `http://www-unix.mcs.anl.gov/AR/eqp/`, 1996–2005.

[Meh03] K. Mehlhorn. The reliable algorithmic software challenge RASC. In R. Klein et al., editors, *Computer Science in Perspective*, volume 2598 of *LNCS*, pages 255–263. Springer, 2003.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[MN90] U. Martin and T. Nipkow. Ordered rewriting and confluence. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNCS*, pages 366–380. Springer, 1990.

[MN98] K. Mehlhorn and S. Näher. From algorithms to working programs: On the use of program checking in LEDA. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1450 of *LNCS*, pages 84–93. Springer, 1998.

[MS00] W. McCune and O. Shumsky. System description: Ivy. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *LNCS*, pages 401–405. Springer, 2000. See: `http://www-unix.mcs.anl.gov/~mccune/acl2/ivy/`.

[New42] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[Nie93a] R. Nieuwenhuis. A new ordering constraint solving method and its applications. Technical Report MPI-I-92-238, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1993.

[Nie93b] R. Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47:65–69, 1993.

[Nie99] R. Nieuwenhuis. Rewrite-based deduction and symbolic constraints (invited paper). In H. Ganzinger, editor, *Proceedings of the International 16th International Conference on Automated Deduction*, volume 1632 of *LNCS*, pages 302–313. Springer, 1999.

[NR95] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[NR02] R. Nieuwenhuis and J.M. Rivero. Practical algorithms for deciding path ordering constraint satisfaction. *Information and Computation*, 178(2):422–440, 2002.

[NRV98] P. Narendran, M. Rusinowitch, and R. Verma. RPO constraint solving is in NP. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *12th Computer Science Logic*, volume 1584 of *LNCS*, pages 385–398. Springer, 1998.

[NW01] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.

[Pet90] G. E. Peterson. Complete sets of reductions with constraints. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNCS*, pages 381–395. Springer, 1990.

[Pla93] D.A. Plaisted. Polynomial time termination and constraint satisfaction tests. In C. Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 405–420. Springer, 1993.

[Plo72] G. D. Plotkin. Building-in equational theories. *Machine Intelligence*, 7(73–90), 1972.

[PP93] A. Pettorossi and M. Proietti. Rules and Strategies for Program Transformation. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 263–304. Springer, 1993.

[Pro98] T. Proebsting. Proebsting's law: Compiler advances double computing power every 18 years. `http://research.microsoft.com/~toddpro/papers/law.htm`, 1998.

[PS81] G.E. Peterson and M.E. Stickel. Complete sets of reduction for some equational theories. *Journal of the Association for Computing Machinery*, 28:233–264, 1981.

[Pug92] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992. Library available at `http://www.cs.umd.edu/projects/omega/`.

[PZ97] D.A. Plaisted and Y. Zhu. Equational reasoning using AC constraints. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 108–113. Morgan Kaufmann, 1997.

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

[Ron03] R. Rondot. Constraintbasierte Vervolständigungstechniken: Ordnungsconstraints. Projektarbeit, Universität Kaiserslautern, Fachbereich Informatik, 2003.

[RSV01] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier, 2001.

[RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of First International Joint Conference on Automated Reasoning*, volume 2083 of *LNCS*, pages 376–380. Springer, 2001.

[RV03] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1–2):101–115, 2003.

[RV04] A. Riazanov and A. Voronkov. Efficient checking of term ordering constraints. In D. Basin and M. Rusinowitch, editors, *Proc. 2nd International Joint Conference on Automated Reasoning*, LNCS, pages 60–74. Springer, 2004.

[RW69] G. A. Robinson and L. T. Wos. Paramodulation and theorem proving in first order theories with equality. *Machine Intelligence*, 4:133–150, 1969.

[S+] G. Sutcliffe et al. The CADE ATP System Competition (CASC). `http://www.tptp.org/CASC`.

[Sch00] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15:111–126, 2002. System's home page: `http://www.eprover.org`.

[SDK+03] A. Stump, A. Deivanayagam, S. Kathol, D. Lingelbach, and D. Schobel. Rogue Decision Procedures. In C. Tinelli and S. Ranise, editors, *1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2003.

[Sha02] N. Shankar. Little engines of proof. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceeding of the International Symposium of Formal Methods Europe*, volume 2391 of *LNCS*, pages 1–20. Springer, 2002.

[Slo05] N.J.A. Sloane. Integer Sequence A000670: Preferential arrangements of n labeled elements; weak orders on n labeled elements. On-Line Encyclopedia of Integer Sequences, `http://www.research.att.com/projects/OEIS?Anum=A000670`, last checked 2005.

[Sny93] W. Snyder. On the complexity of recursive path orderings. *Information Processing Letters*, 46:257–262, 1993.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. See `http://www.tptp.org`.

[Ste94] J. Steinbach. *Termination of Rewriting.* PhD thesis, Universität Kaiserslautern, 1994.

[Tam97] T. Tammet. Gandalf. *Journal Automated Reasoning*, 18(2):199–204, 1997.

[Tür03] T. Türk. Constraintbasierte Vervolständigungstechniken: Gleichheitsconstraints. Projektarbeit, Universität Kaiserslautern, Fachbereich Informatik, 2003.

[Vor01] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction (invited talk). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning*, volume 2083 of *LNCS*, pages 13–28. Springer, 2001.

[Wad92] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[WB97] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 275–279. Springer, 2002. System's home page: `http://spass.mpi-sb.mpg.de/`.

[Wei99] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *LNCS*, pages 314–328. Springer, 1999.

[Wei01] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[Wol02] S. Wolfram. *A New Kind of Science.* Wolfram Media, 2002.

[ZK90] H. Zhang and D. Kapur. Unnecessary inferences in associative-commutative completion procedures. *Mathematical Systems Theory*, 23(3):175–206, 1990.

[ZSM05] T. Zhang, H. B. Sipma, and Z. Manna. The decidability of the first-order theory of Knuth-Bendix order. Submitted, 2005.

Curriculum Vitae

Persönliche Daten

Name	Bernd Löchner
Geburtsdatum	26.10.1967
Geburtsort	Ludwigshafen/Rhein
Familienstand	verheiratet, ein Kind
Staatsangehörigkeit	deutsch

Schulbildung

1974–1978	Grundschule Schillerschule, Haßloch/Pfalz
1978–1987	Leibniz Gymnasium, Neustadt/Weinstr. Abschluß: Abitur

Grundwehrdienst

1987–1988	Clausthal-Zellerfeld und Andernach

Hochschulstudium

1988–1996	Informatik mit Nebenfach Physik Universität Kaiserslautern Abschluß: Diplom-Informatiker

Berufstätigkeit

1996–2005	Wissenschaftlicher Mitarbeiter Fachbereich Informatik Universität Kaiserslautern