Flexible Heuristic Control for Combining Automation and User-Interaction in Inductive Theorem Proving

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

Dipl.-Inform. Tobias Schmidt-Samoa

Promotionskommission:

Vorsitzender:	Prof. Dr. Klaus Schneider
Berichterstatter:	Prof. Dr. Jürgen Avenhaus
	Prof. Dr. Klaus Madlener
Dekan:	Prof. Dr. Reinhard Gotzhein

Datum der wissenschaftlichen Aussprache: 16. Juni 2006

D 386

To Katja

Acknowledgements

I would like to thank Jürgen Avenhaus for giving me the opportunity to work in his research group and supporting me in finishing this thesis. His comments and suggestions helped to improve the presentation and readability of this thesis a lot. Furthermore, I would like to thank Klaus Madlener for spending much time and effort in refereeing this thesis resulting in improvements in the final version of the thesis, and Klaus Schneider for chairing its defense.

I am much indebted to my colleagues and friends Claus-Peter Wirth and Bernd Löchner for many fruitful discussions and their friendly advice. Perhaps this thesis would have never been finished without Claus-Peter's enthusiasm, encouragement and support, in particular, his patience and effort during proof-reading this thesis and his suggestions for improving it. I owe more to Claus-Peter than I can express here. Special thanks to Bernd for initiating the most complex of my case studies.

This thesis could not have be done without previous work supervised by Ulrich Kühler and realized by Christian Embacher, Jürgen Schumacher, and Christof Sprenger which led to the first version of QuodLibet. With their diploma theses, René Rondot and Markus Kaiser helped me with the realization of my ideas. René did a great job working out some of the technical details for the integration of the decision procedure for linear arithmetic into QuodLibet.

As a student as well as a scientific researcher I got to know many people in Jürgen Avenhaus' and Klaus Madlener's research groups. I would like to thank them all—namely, Martin Anlauf, Arnim Buch, Thomas Deiß, Jörg Denzinger, Robert Eschbach, Dirk Fuchs, Marc Fuchs, Matthias Fuchs, Thomas Hillenbrand, Christoph Kögl, Birgit Reinert, Wilson Castro Rojas, Andrea Sattler-Klein, Stephan Schulz, Vladimir Támara, Dirk Zeckzer, and those that I have forgotten accidently—for the pleasant working atmosphere and the things I have learned from them. Special thanks to Martin Kronenburg for supervising my diploma thesis; to Bernd Strieder—who has always been a reliable fellow and a valued friend—and Tobias Wahl for their good collaboration regarding system administration and teaching activities, respectively; and to Bernhard Gramlich for his interest in and support of my work.

This thesis is dedicated to my lovely and beloved wife Katja. She always encouraged me and facilitated my job to the best I can imagine. Therefore, this thesis would not have been completed without her. Thank you for all your love and patience, Katja. Last but not least, I would like to thank my parents for their continuous support through all the years which I appreciate very much.

Abstract

The validity of formulas w.r.t. a specification over first-order logic with a semantics based on all models is semi-decidable. Therefore, we may implement a proof procedure which finds a proof for every valid formula fully automatically. But this semantics often lacks intuition: Some pathological models such as the trivial model may produce unexpected results w.r.t. validity. Instead, we may consider just a class of special models, for instance, the class of all data models. Proofs are then performed using induction. But, inductive validity is not semi-decidable—even for first-order logic. This theoretical drawback manifests itself in practical limitations: There are theorems that cannot be proved by induction directly but only generalizations can be proved. For their definition, we may have to extend the specification. Therefore, we cannot expect to prove interesting theorems fully automatically. Instead, we have to support user-interaction in a suitable way.

In this thesis, we aim at developing techniques that enhance automatic proof control of (inductive) theorem provers and that enable user-interaction in a suitable way. We integrate our new proof techniques into the inductive theorem prover QuodLibet and validate them with various case studies. Essentially, we introduce the following three proof techniques:

1. We integrate a decision procedure for linear arithmetic into QuodLibet in a *close* way by defining new inference rules that perform the elementary steps of the decision procedure. This allows us to implement well-known heuristics for automatic proof control. Furthermore, we are able to provide special purpose tactics that support the manual speculation of lemmas if a proof attempt gets stuck. The integration improves the ability of the theorem prover to prove theorems automatically as well as its efficiency. Our approach is competitive with other approaches regarding efficiency; it provides advantages regarding the speculation of lemmas.

2. The automatic proof control searches for a proof by applying inference rules. The search space is not only infinite, but grows dramatically with the depth of the search. In contrast to this, checking and analyzing performed proofs is very efficient. As the search space also has a high redundancy, it is reasonable to *reuse* subproofs found during proof search. We define new notions for the *contribution* of proof steps to a proof. These notions enable the derivation of pruned proofs and the identification of superfluous subformulas in theorems. A proof may be reused in two ways: *upward propagation* prunes a proof by eliminating superfluous proof steps; *sideward reuse* closes an open proof obligation by replaying an already found proof.

3. For interactive theorem provers, it is essential not only to prove automatically as many lemmas as possible but also to restrict proof search in such a way that the proof process stops within a reasonable amount of time. We introduce different *markings* in the goals to be proved and the lemmas to be applied to restrict proof search in a flexible way: With a *forbidden* marking, we can simulate well-known approaches for applying conditional lemmas. A *mandatory* marking provides a new heuristics which is inspired by local contribution of proof steps. With *obligatory* and *generous* markings, we can fine-tune the degree of efficiency and extent of proof search manually.

With an elaborate case study, we show the benefits of the different techniques, in particular the synergetic effect of their combination.

Zusammenfassung

Die Gültigkeit prädikatenlogischer Formeln erster Stufe ist semi-entscheidbar, sofern man bei der Semantik alle Modelle zugrundelegt. Daher ist es möglich ein Beweisverfahren zu implementieren, das für jede gültige Formel einen Beweis vollautomatisch findet. Aber eine Semantik, die alle Modelle berücksichtigt, entspricht oft nicht der Intuition: Einige pathologische Modelle, wie z. B. das triviale Modell, führen zu unerwarteten Resultaten bzgl. der Gültigkeit. In diesem Fall können wir die Klasse der betrachteten Modelle einschränken, z. B. auf die Klasse aller Datenmodelle. Beweise werden in dieser Klasse mit Hilfe von Induktion geführt. Die resultierende induktive Gültigkeit ist jedoch nicht mehr semi-entscheidbar—nicht einmal für die Prädikatenlogik erster Stufe. Dies hat praktische Auswirkungen: Es gibt Theoreme, die sich nicht direkt durch Induktion zeigen lassen. Statt dessen müssen Generalisierungen betrachtet werden. Um diese definieren zu können, ist evtl. eine Erweiterung der Spezifikation nötig. Daher können wir nicht erwarten, interessante Theoreme vollautomatisch beweisen zu können. Statt dessen müssen wir manuelle Interaktionen in geeigneter Weise unterstützen.

In dieser Arbeit entwickeln wir Techniken, die die automatisierte Beweissteuerung von (induktiven) Theorembeweisern verbessern und die manuelle Interaktionen in geeigneter Weise unterstützen. Wir integrieren unsere neuen Beweistechniken in den induktiven Theorembeweiser QuodLibet und validieren sie mit Hilfe diverser Fallstudien. Im Wesentlichen führen wir die folgenden drei neuen Beweistechniken ein:

1. Wir integrieren eine Entscheidungsprozedur für lineare Arithmetik in QuodLibet auf *enge* Weise, indem wir neue Inferenzregeln definieren, die den elementaren Schritten der Entscheidungsprozedur entsprechen. Auf diese Weise können wir alle Heuristiken implementieren, die für eine automatisierte Beweissteuerung bekannt sind. Außerdem können wir spezielle Taktiken zur Verfügung stellen, die im Fall erfolgloser Beweisversuche die manuelle Spekulation von Lemmata unterstützen. Die Integration verbessert sowohl die Effektivität als auch die Effizienz der automatisierten Beweissuche. Unser Ansatz ist bzgl. der Effizienz mit anderen Ansätzen vergleichbar; er bietet jedoch zusätzliche Vorteile beim Spekulieren von Lemmata.

2. Die automatisierte Beweissteuerung sucht Beweise durch Anwenden von Inferenzregeln. Der Suchraum ist nicht nur unendlich, sondern wächst dramatisch in Abhängigkeit von der Tiefe der Suche. Im Gegensatz zur Suche neuer Beweise kann die Analyse von gefundenden Beweise sehr effizient durchgeführt werden. Da der Suchraum eine hohe Redundanz aufweist, ist es sinnvoll gefundene Teilbeweise *wiederzuverwenden*. Wir führen neue Begriffe ein, die den *Beitrag* eines Beweisschritts zu einem Beweis festlegen. Diese Begriffe erlauben das Bereinigen von Beweisen und die Identifikation von unnötigen Teilformeln, die zum Beweis eines Theorems nichts beitragen. Ein Beweis kann auf zwei Arten wiederverwendet werden: Die *Aufwärtspropagierung* bereinigt einen Beweis, indem überflüssige Beweisschritte entfernt werden; die *Seitwärtswiederverwendung* überträgt einen alten Beweis auf ein offenes Beweisziel.

3. Für interaktive Beweiser ist es wichtig, nicht nur so viele Lemmata wie möglich automatisch zu beweisen, sondern auch die Beweissuche derart einzuschränken, dass sie in vernünftiger Zeit terminiert. Wir führen verschiedene *Markierungen* ein, um die

Beweissuche in flexibler Weise zu steuern—zum einen auf den Zielen, die bewiesen werden sollen, und zum anderen auf den Lemmata, die angewendet werden dürfen: Mit einer *verbotenen* Markierung können wir bekannte Ansätze simulieren, die die Anwendung von bedingten Lemmata steuern. Eine *verpflichtende* Markierung führt zu einer neuen Heuristik, die auf dem lokalen Beitrag von Beweisschritten beruht. Mit *obligatorischen* und *großzügigen* Markierungen schließlich können wir die Effizienz und den Umfang der Beweissuche manuell steuern.

Anhand einer umfassenden Fallstudie zeigen wir die Vorteile unserer verschiedenen Techniken auf, insbesondere den synergetischen Effekt, der aus ihrer Kombination resultiert.

Contents

Chapter 1

Introduction

1.1 Motivation

If we want to establish "properties" of an "environment", we first have to formalize the *application domain.* This is usually done within some kind of logic. A *logic* fixes the

syntax, i.e. how do we construct *logic objects*, e.g. *terms* and *formulas*, from basic elements; and

semantics, i.e. which formulas are consequences of basic formulas.

The formalization of the application domain is given by a specification. A *specification* consists of a signature and axioms. The *signature* defines the basic elements, whereas the *axioms* define the basic formulas. Typically, the semantics of a logic is given by a class of *models* of the specification, i.e. algebras of the signature that fulfill all the axioms. Whereas the environment is described by the axioms, the properties are given by additional formulas called *theorems* (or *lemmas*). To establish a property of the environment, we have to prove that the according theorem is *valid* in the specification, i.e. that it is a consequence of the axioms. Instead of using semantic arguments, we prove theorems with sound logic calculi. Roughly speaking, a *logic calculus* for a logic consists of inference rules that enable the derivation of new formulas (*conclusions*) from a given set of formulas (*premises*). The logic calculus is *sound* if every derived formula is valid. The logic calculus is *complete* if we can derive every valid formula.

Depending on the logic, the balance between

expressiveness, i.e. which relationships can be expressed;

automation, i.e. which theorems can be proved automatically; and

intuition, i.e. is the validity of the theorems as we expect;

varies. Propositional logic has a high degree of automation as it is decidable, but lacks expressiveness. For automated theorem proving, (clausal) first-order logic with semantics based on all models of the specification is often considered as a suitable compromise between expressiveness and automation since it is semi-decidable. Therefore, there exist sound and complete logic calculi for first-order logic with semantics based on all models. This entails that we can enumerate all valid formulas of a specification. Theoretically (without regarding space and time limitations), this yields a proof procedure which finds a proof for every valid first-order formula of the specification.

But even first-order logic with semantics based on all models is often inappropriate because it lacks expressiveness or intuition. In the first case, we may switch to higher-order logic. The latter case is caused by the fact that the user often has a special model in mind when he formalizes an application domain, e.g. when he specifies an *abstract data type* and proves properties of it. Thus, the class of all models does not match his intuition. Some pathological models such as the trivial model may produce unexpected results w.r.t. the validity of theorems. In this case, we have to restrict the class of models considered for the semantics of a specification. To prove theorems in the *initial model*, the class of all *term-generated models* or all *data models* (cf. Chapter 2) of a specification, we have to use *induction* on (constructor) terms. Therefore, these classes provide different alternatives for defining the *inductive validity* of theorems (cf. [AM97, Wir97]). Whereas *deductive theorem proving* is concerned with the validity in all models, *inductive theorem proving* deals with inductive validity. Neither the validity in higher-order logic nor the inductive validity in first-order logic are semi-decidable.

In this thesis, we are concerned with inductive theorem proving in clausal first-order logic. The theoretical drawback—namely, that the inductive validity of theorems is not even semi-decidable—manifests itself in practical limitations: In contrast to deductive theorem proving, applications of Gentzen's Cut rule cannot be eliminated in inductive proofs [Kre65]. Thus, we have to guess intermediate formulas to prove theorems. In practice, there are theorems that cannot be proved by induction directly but only generalizations of them can be proved. To formulate the required auxiliary lemmas, we may even have to extend the signature and the axioms of the specification (cf. Chapter 8). Therefore, we cannot expect to prove interesting theorems fully automatically. This leads to the following insight in [Wir04, page 10]:

> 'Successful application of an inductive theorem prover in "real-life" domains requires a knowledgeable human user who can interact with the system at various levels of abstraction.'

In this thesis, we aim at developing techniques that enhance automatic proof control of (inductive) theorem provers and that enable user-interaction in a suitable way. The underlying ideas of our techniques are neither limited to a special theorem prover nor to inductive theorem proving. Most of the techniques, however, display their full power only in conjunction with user-interaction as required in inductive theorem proving. To concretize our techniques enhancing readability and understanding, and to validate the usefulness of our techniques with case studies, we integrate them into an inductive theorem prover. For this, we choose QuodLibet [AKSSW03, Küh00]. QuodLibet is not as efficiently implemented as the industrial-strength inductive theorem prover `ACL2` [KMM00] for program verification. Instead, QuodLibet was designed with a strong emphasis on both, automation as well as user-orientation. Therefore, it provides the following advantages for the integration of our new techniques:

Strict separation between logic engine and proof control: The inference system provides *elementary* proof steps which are oriented towards human proof techniques. Proofs can be automated combining elementary proof steps in *tactics* written in an adapted imperative programming language QML.

The inference rules are widely applicable. Basically, their applicability is solely restricted to guarantee soundness and safeness properties of the inference rules. In contrast to other approaches, the applicability is not restricted by heuristics.[1] Therefore, our inference rules are suitable for simulating many different approaches for proof control. The currently considered proof control is then responsible for restricting the applications heuristically.

Easy extensibility: Both, the inference system and the proof control can be extended easily. Local properties of the inference rules guarantee the soundness of the extension.

Explicit representation of proofs with proof state graphs: This representation is particularly useful if a proof attempt gets stuck. Proof state trees provide the user with information about the failed proof attempt. In a uniform and flexible way, the user may provide a hint for continuing the proof attempt e.g. by applying an inference rule or calling a tactic.

Furthermore, the maintenance of the dependencies between proof attempts of (different) lemmas in a proof state graph enables the lazy generation of induction hypotheses, the application of yet unproved lemmas, and the usage of multiple proof attempts. These techniques are advantageous for performing complicated proofs.

Suitable semantics for partially defined operators: In our case studies, many operators are only partially defined. QuodLibet provides welldefined semantics for specifying these operators in a natural way.

Our new techniques are intended to support complicated proofs as e.g. those based on mutual induction. Essentially, we introduce the following three techniques to combine automation and user-interaction:

1. Inspired by the seminal work in [BM88b], we integrate a decision procedure for linear arithmetic into QuodLibet. Our form of *integration* is very *close* as we define new inference rules that perform the elementary steps of the decision procedure. This allows us to implement well-known heuristics for automatic proof control with tactics such as the *augmentation* technique. Furthermore, we are able to provide special purpose tactics that support the manual *speculation* of lemmas if a proof attempt gets stuck. The integration improves the ability of the theorem prover to prove theorems automatically as well as its efficiency. In spite of the close integration, our approach is competitive with other approaches known from the literature regarding efficiency; it provides additional advantages regarding the manual speculation of auxiliary lemmas.

2. Primarily, the automatic proof control *searches* for a proof by applying inference rules. The search space is not only infinite, but grows dramatically with the depth of the search. In contrast to this, checking and analyzing performed proofs is very efficient.

[1] Rewriting may, for instance, be restricted by wellfounded orders.

As the search space also has a high redundancy, it is reasonable to *reuse* subproofs found during proof search.

We define new notions for the *contribution* of proof steps to a proof. These notions enable the derivation of pruned proofs and the identification of superfluous subformulas in theorems enhancing the reusability of proofs. A proof may be reused in two ways: *upward propagation* propagates a proof upwards in the proof state tree eliminating superfluous proof steps and the corresponding proof obligations; *sideward reuse* propagates a proof sidewards in the proof state tree closing a formerly open proof obligation.

3. In particular for interactive theorem provers, it is essential not only to prove automatically as many lemmas as possible but also to restrict proof search in a suitable way such that the proof process stops within a reasonable amount of time. We introduce different *markings* to restrict proof search in a flexible way: A *forbidden* marking can be used for simulating well-known approaches in the literature for applying conditional lemmas. A *mandatory* marking provides a new heuristics which is inspired by a local form of contribution of proof steps. With *obligatory* and *generous* markings in lemmas, the user can fine-tune the degree of *efficiency* and *extent* of proof search manually.

1.2 Overview of the Thesis

We start with two introductory chapters about the inductive theorem prover QuodLibet and the basics of our automatic proof control.

More precisely, in Chapter 2, we describe the logic of QuodLibet and its architecture. The basics presented in this chapter are required throughout this thesis. On the one hand, QuodLibet has initiated the development of the new proof techniques. On the other hand, we perform case studies within QuodLibet to validate the new proof techniques. In particular, the inference rules of QuodLibet (cf. Section 2.2.2) and the tactic-based proof control (cf. Section 2.3.1) are essential for our new proof techniques.

In Chapter 3, we present our principle approach for the automation of inductive theorem proving. We consider the integration of induction schemes into the proof process as well as the top-level organization of proof search. For the integration of induction schemes, there are at least three different approaches known from the literature: proof by consistency (cf. Section 3.1.1), explicit induction (cf. Section 3.1.2), and descente infinie (cf. Section 3.1.3). Due to its support of user-interaction, our proof process is based on descente infinie. Our top-level proof control is inspired by the simple waterfall model used in `NQTHM` [BM88a] and `ACL2` [KMM00]. We refine their simple waterfall to a more flexible one. This results in a table-based proof control described in Section 3.2.2 which allows us to integrate new proof techniques easily.

In the following four chapters, we present the main contributions of this thesis: the close integration of a decision procedure for linear arithmetic into QuodLibet; the notion of contribution for adaptable inference systems and its applications—guiding proof search with markings and reusing contributing proofs.

The integration of decision procedures into theorem provers has been studied for decades. It aims at enhancing the scope of the theorem provers—the lemmas that can be proved automatically—as well as the efficiency of proof search. In Chapter 4, we present a novel approach which performs the integration closely. We derive new inference rules for QuodLibet which represent the elementary steps of the decision procedure. The applications of the inference rules may be automated in different ways which are more suited for automation on the one hand, and for the manual speculation of auxiliary lemmas on the other. In Section 4.3, we validate our new approach with various case studies: We compare the old proof control before the integration with the new proof control after the integration; we provide evidence that our new approach is competitive with other approaches known from the literature; and we illustrate the benefits of our new approach w.r.t. the manual speculation of auxiliary lemmas. We have presented a preliminary version of this work in [SS06a].

Both—guiding proof search with markings and reusing proofs—depend on partitioning a goal into principal part, cut-off part, and context w.r.t. the inference rule applied. Both approaches are not restricted to QuodLibet but may be applied to other reductive inference systems working on proof state trees as well. The common basics are captured in the notion of adaptable inference system in Chapter 5. This notion allows us to extract the essence of a proof w.r.t. its contribution to the proof.

Marking techniques provide a flexible and uniform way for guiding proof search. We distinguish two kinds of markings: Markings in the goals to be proved and markings in the lemmas to be applied. Markings in the goals guide proof search in two steps. Firstly, in a fixed way, we define restrictions on the proof steps that can be performed w.r.t. the markings in the goals and the partitioning of the goal into principal part, cut-off part and context. Secondly, we use heuristics to inherit the markings to the new subgoals. With a forbidden marking, we can model previous approaches known from the literature such as Contextual and Case Rewriting. With a mandatory marking, we introduce a novel approach for guiding proof search which is based on local contribution of proof steps. With markings in the lemmas, we can influence the efficiency and the extent of proof search in a flexible way manually. Primarily, obligatory markings are intended to enhance the efficiency, whereas generous markings enhance the extent. However, enhancing the extent may also improve efficiency. In Chapter 6, we present our flexible framework for guiding proof search in detail. A self-contained version of this chapter will appear in [SS06b].

Instead of guiding proof search, we may also utilize the contribution of proof steps for improving and reusing performed proofs. In Chapter 7, we present two different reuse mechanisms: upward propagation and sideward reuse. The first one eliminates superfluous proof steps. In doing so, we may also eliminate open proof obligations. The second one allows us to prove an open subgoal without proof search and backtracking but with adapting a proof previously performed successfully. These reuse mechanisms replace proof search to some extent by proof reuse, resulting in a more efficient proof process.

We continue this thesis with a comprehensive case study about the lexicographic path order LPO in Chapter 8. This case study is challenging as it contains function symbols which are defined by mutual recursion. This calls for proofs based on mutual induction which causes several difficulties in speculating auxiliary lemmas for all dependent operators, in performing appropriate inductive case splits, in applying suitable induction hypotheses, in finding suitable wellfounded induction orders, and in proving the corresponding order constraints. Our new proof techniques help us in this endeavor. In this chapter, we sketch our proof engineering process as well as the resulting proof script. We illustrate the synergetic effect of our new proof techniques which results e.g. from the combination of generous markings and upward propagation. Furthermore, we point out directions of further research.

Finally, we conclude this thesis in Chapter 9.

Chapter 2

QuodLibet: The Logic and the Architecture

In this thesis, we use the inductive theorem prover QuodLibet to illustrate our new techniques for automating inductive proofs and to validate them with case studies. QuodLibet is an *equality-based* inductive theorem prover for *clausal first-order logic* with implicitly *universally* quantified variables. It admits *partial* definitions of operators over *free constructors* using (possibly *non-terminating*) conditional equations as well as *constructor*, *destructor*, and *mutual* recursion. Therefore, it is well suited for complex specifications of *abstract data types*. *Inductive validity* is defined as validity in the class of so-called *data models*, the models that do not equalize any different constructor ground terms. Proofs can be performed by induction on constructor ground terms. QuodLibet permits the explicit application of a lemma as induction hypothesis resulting in an additional order obligation to guarantee the wellfoundedness of the induction order. This enables the modeling of different inductive proof processes such as *explicit induction* and *descente infinie* (cf. Section 3.1). Proofs are based on a *sequent calculus* and represented by *proof state graphs*. Inference rules may be applied *manually* or called *automatically* with *tactics* written in an adapted imperative programming language QML. User-interaction may be performed using a text-based or a graphical user-interface called XQL.

In this chapter, we summarize the foundations of QuodLibet as they are relevant for this thesis. Particularly, this chapter answers the following questions:

- How do we formalize specifications with QuodLibet?
- Which properties can be expressed with QuodLibet?
- What is the semantics of a specification?
- How do we prove lemmas with QuodLibet?

A detailed description including motivation of the design decisions can be found in [Küh00] based on theoretical work in [KW97, Wir97]. An extension to existentially quantified variables and arbitrary two-valued logics is described in [Wir04]. Further information about XQL and QML can be found in [SK97] and [SS97], respectively.

In Section 2.1, we summarize general basic notions required for this thesis. In Section 2.2, we present the logic of QuodLibet—consisting of the specification language (2.2.1) and the inference system (2.2.2) for performing proofs—and illustrate the given notions with an example (2.2.3). The architecture of QuodLibet which is structured in three levels is described in Section 2.3. Of special interest for the automation of proofs is QML presented in 2.3.1.

2.1 Basic Notions

We expect the reader to be familiar with fundamental concepts of algebraic specifications and term rewriting. In this section, we summarize some basic notions. Further details can be found in [AM90, Ave95, BN98, Wir90].

Signatures. A many-sorted *signature* $\Sigma = (S, F, \alpha)$ consists of a set S of *sorts*, a set F of *function symbols* and an *arity function* $\alpha : F \to S^+$. The arity function α assigns to each function symbol f argument sorts $s_1 \dots s_n$ and a result sort s. Instead of $\alpha(f) = s_1 \dots s_n s$, we also write $f : s_1, \dots, s_n \to s$. Let $V = (V_s)_{s \in S}$ be an S-sorted family of infinite disjoint sets V_s of *variable symbols* for each sort s with $V \cap F = \varnothing$.

Terms. The set of *wellformed terms* $\mathrm{Term}_s(F, V)$ of sort s is recursively defined as the smallest set with

- $V_s \subseteq \mathrm{Term}_s(F, V)$ and
- $f(t_1, \dots, t_n) \in \mathrm{Term}_s(F, V)$ if f is a function symbol with $\alpha(f) = s_1 \dots s_n s$ and $t_i \in \mathrm{Term}_{s_i}(F, V)$ for each $i \in \{1, \dots, n\}$.

A term that only consists of a *constant* c, i.e. a function symbol without argument sorts, is often denoted by c instead of $c()$. The set of all wellformed terms is denoted by $\mathrm{Term}(F, V)$. With $\mathrm{Term}(F)$, we denote the set of wellformed *ground terms*, i.e. terms that do not contain any variable.

The *length* $|t|$ of a term t is recursively defined as follows:

- $|t| = 1$ if $t \in V$ and
- $|t| = 1 + \sum_{i=1}^{n} |t_i|$ if $t \equiv f(t_1, \dots, t_n)$.

The set of *variables* $V(t)$ of a term t is defined as follows:

- $V(t) = \{t\}$ if $t \in V$ and
- $V(t) = \bigcup_{i=1}^{n} V(t_i)$ if $t \equiv f(t_1, \dots, t_n)$.

The number of *occurrences* $|t|_x$ of a variable x in a term t is recursively defined as

- $|t|_x = \begin{Bmatrix} 1 & \text{if } t \equiv x \\ 0 & \text{if } t \not\equiv x \end{Bmatrix}$ if $t \in V$ and
- $|t|_x = \sum_{i=1}^{n} |t_i|_x$ if $t \equiv f(t_1, \ldots, t_n)$.

A term t is called *linear* if $|t|_x \leq 1$ for each $x \in V(t)$.

The *top-level* symbol $\text{top}(t)$ of a term t is defined as

- $\text{top}(t) = t$ if $t \in V$ and
- $\text{top}(t) = f$ if $t \equiv f(t_1, \ldots, t_n)$.

Positions. The set $Pos(t)$ of *positions* of a term t is the smallest set with

- the empty position $\varepsilon \in Pos(t)$ and
- $i.p \in Pos(t)$ if $t \equiv f(t_1, \ldots, t_n)$, $i \in \{1, \ldots, n\}$ and $p \in Pos(t_i)$.

On positions we define a *prefix relation* $\leq_{\text{prefix}}$ as smallest relation with

- $\varepsilon \leq_{\text{prefix}} p'$ for each position p' and
- $i.p \leq_{\text{prefix}} i.p'$ if $p \leq_{\text{prefix}} p'$.

A position p is *minimal* in a set P of positions if $p \in P$ and for each $p' \in P$, if $p' \leq_{\text{prefix}} p$ then $p' \equiv p$.

The *depth* $|p|$ of a position p is its length, i.e.

- $|p| = 0$ if $p \equiv \varepsilon$ and
- $|p| = 1 + |p'|$ if $p \equiv i.p'$.

The *prefix* $\text{prefix}(p, d)$ of position p with depth at most $d \in \mathbb{N} \cup \{\omega\}$ is

- $\text{prefix}(p, d) = p$ if $|p| \leq d$ and
- $\text{prefix}(p, d) = p'$ with $p' \leq_{\text{prefix}} p$ and $|p'| = d$ if $|p| > d$

Subterms. The *subterm* t/p of term t at position $p \in Pos(t)$ is recursively defined as follows:

- $t/p \equiv t$ if $p \equiv \varepsilon$ and
- $t/p \equiv t_i/p'$ if $t \equiv f(t_1, \ldots, t_n)$ and $p \equiv i.p'$.

With $t[u]_p$ we denote the *replacement* of the subterm of t at position $p \in Pos(t)$ with u defined as:

- $t[u]_p \equiv u$ if $p \equiv \varepsilon$ and
- $t[u]_p \equiv f(t_1, \ldots, t_{i-1}, t_i[u]_{p'}, t_{i+1}, \ldots, t_n)$ if $t \equiv f(t_1, \ldots, t_n)$ and $p \equiv i.p'$.

Substitutions. A *substitution* is a sort-preserving function $\sigma : V \longrightarrow \mathrm{Term}(F,V)$, i.e. $\sigma(x) \in \mathrm{Term}_s(F,V)$ for each $x \in V_s$, with finite domain $dom(\sigma) = \{x \in V \mid \sigma(x) \not\equiv x\}$. It is extended to a function on $\mathrm{Term}(F,V)$ by $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.[1] The *composition* $\sigma\tau$ of two substitutions σ and τ is defined as $x(\sigma\tau) \equiv (x\sigma)\tau$.

A *match* from term l to term t is a substitution σ with $l\sigma \equiv t$. Note that, if there is a match from l to t, it is unique on $V(l)$.

Terms u and v are *unifiable* if there exists a *unifier* σ, i.e. a substitution with $u\sigma \equiv v\sigma$. A *most general unifier* of u and v is a unifier σ of u and v such that, for every unifier μ of u and v, there exists a substitution τ with $\mu = \sigma\tau$. The most general unifier of u and v is unique on $V(u) \cup V(v)$ up to a variable renaming. We denote it by $mgu(u,v)$ if it exists.

Σ-algebras. A Σ-*algebra* $\mathcal{A} = (A, F^{\mathcal{A}})$ for a signature $\Sigma = (S, F, \alpha)$ is defined by

- a universe $A = (A_s)_{s \in S}$ where A_s is a non-empty set for each $s \in S$ and
- a set of functions $F^{\mathcal{A}} = (f^{\mathcal{A}})_{f \in F}$ with $f^{\mathcal{A}} : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ if $\alpha(f) = s_1 \ldots s_n s$.

A *valuation* is a function $\varphi : V \rightarrow A$ with $\varphi(x) \in A_s$ for each $x \in V_s$. The valuation φ for an algebra $\mathcal{A}$ is extended to terms by a function $\mathrm{eval}_\varphi^{\mathcal{A}}$ defined as

- $\mathrm{eval}_\varphi^{\mathcal{A}}(t) \equiv \varphi(t)$ if $t \in V$ and
- $\mathrm{eval}_\varphi^{\mathcal{A}}(t) \equiv f^{\mathcal{A}}(\mathrm{eval}_\varphi^{\mathcal{A}}(t_1), \ldots, \mathrm{eval}_\varphi^{\mathcal{A}}(t_n))$ if $t \equiv f(t_1, \ldots, t_n)$.

For each ground term $t \in \mathrm{Term}(F)$, the evaluation $\mathrm{eval}_\varphi^{\mathcal{A}}(t)$ results in the same element of the universe regardless of the valuation φ. Therefore, we also denote this element by $\mathrm{eval}^{\mathcal{A}}(t)$ or $t^{\mathcal{A}}$.

Rewrite relations. A *rewrite relation* $\longrightarrow$ is a binary relation on a set A. Let $\longleftarrow$ be its *reverse* relation, $\longleftrightarrow$ its *symmetric* closure, $\xrightarrow{+}$ its *transitive* closure, and $\xrightarrow{*}$ its *reflexive-transitive* closure. Its *joinability* relation $\downarrow$ is defined by $\downarrow\, = \xrightarrow{*} \circ \xleftarrow{*}$.

The rewrite relation is *confluent* if $\xleftarrow{*} \circ \xrightarrow{*} \subseteq \downarrow$.

The rewrite relation is *terminating* if there does not exist an infinite sequence $(a_i)_{i \in \mathbb{N}}$ with $a_i \longrightarrow a_{i+1}$ for each $i \in \mathbb{N}$.

[1] We use postfix notation to denote the application $t\sigma$ of a substitution σ to a term t.

(Quasi-)orders. A binary relation $\lesssim$ is a *quasiorder* if it is reflexive and transitive. It is an *order* if it is additionally antisymmetric. Let $< \; = \; \lesssim - \gtrsim$ be the *strict part* of $\lesssim$ and $\sim \; = \; \lesssim \cap \gtrsim$ be the *equivalent part* of $\lesssim$. The (quasi-)order $\lesssim$ is *wellfounded* if $>$ is terminating.

For a set A, let A^* be the set of *finite sequences* with elements in A. Let $\lesssim$ be a (quasi-)order on A. The *lexicographic* order $\lesssim^{\mathrm{lex}}$ on A^* induced by $\lesssim$ is defined as follows: $a_1, \ldots, a_n \lesssim^{\mathrm{lex}} b_1, \ldots, b_m$ iff

- $n = 0$ or
- $n > 0$, $m > 0$ and $a_1 < b_1$ or
- $n > 0$, $m > 0$, $a_1 \sim b_1$ and $a_2, \ldots, a_n \lesssim^{\mathrm{lex}} b_2, \ldots, b_m$.

In general, the lexicographic order $\lesssim^{\mathrm{lex}}$ induced by $\lesssim$ is not wellfounded even if $\lesssim$ is wellfounded. But if we consider sequences up to a fixed length $k_0 \in \mathbb{N}$ only, this holds true, i.e. $\lesssim^{\mathrm{lex}}$ is wellfounded if $\lesssim$ is wellfounded.

Operations on sets and multisets. Both, sets and multisets may be used to represent *clauses*, i.e. disjunctively combined literals. For a uniform treatment, we define the following operations on sets and multisets: A subset relation $\subseteq$, a union $\cup$, an intersection $\cap$, a sum $+$, a difference $-$ operation, and a partition relation $\uplus$.

For sets, we define $\subseteq$ as the subset relation on sets, $\cup$ and $+$ as the union, $\cap$ as the intersection, $-$ as the difference of sets, and $A = A_1 \uplus \cdots \uplus A_m$ as the usual partition relation on sets, i.e. $A_1 \uplus \cdots \uplus A_m$ is a partition of set A if $A = A_1 \cup \cdots \cup A_m$ and A_i are pairwise disjoint sets.

A *multiset* $M(A)$ for set A is a function $M : A \to \mathbb{N}$ such that $\{x \in A \mid M(x) > 0\}$ is finite. We define the operations on multisets as follows:

- $M_1 \subseteq M_2$ iff $M_1(x) \leq M_2(x)$ for all $x \in A$;
- $(M_1 \cup M_2)(x) = \max(M_1(x), M_2(x))$ for all $x \in A$;
- $(M_1 \cap M_2)(x) = \min(M_1(x), M_2(x))$ for all $x \in A$;
- $(M_1 + M_2)(x) = M_1(x) + M_2(x)$ for all $x \in A$;
- $(M_1 - M_2)(x) = \max(0, M_1(x) - M_2(x))$ for all $x \in A$;
- $M = M_1 \uplus \cdots \uplus M_m$ iff $M = M_1 + \cdots + M_m$.

Let $\lesssim$ be a (quasi-)order. The *multiset extension* $\ll$ on $M(A)$ induced by $\lesssim$ is defined as follows: $M_1 \ll M_2$ iff

- there exist $X, Y \in M(A)$ with $\varnothing \neq X \subseteq M_2$, $M_1 = (M_2 - X) + Y$ and for each $y \in Y$ there exists an $x \in X$ with $y < x$.

The multiset extension $\ll$ is wellfounded if $\lesssim$ is wellfounded.

2.2 The Logic

In this section, we summarize the logic of the inductive theorem prover QuodLibet. Since its inference system is based on *free constructors*, we consider only this special case. A more general approach is described in [Küh00].

2.2.1 The Specification Language

2.2.1.1 Syntax

In comparison to the basic notions in Section 2.1, QuodLibet allows for the modeling of partial functions. Therefore, the function symbols of a signature in QuodLibet are divided into constructors and defined function symbols. Let $\Sigma = (S, F, \alpha)$ be a signature. The user has to identify a set of *constructors* $C \subseteq F$ in such a way that the signature $\Sigma^C = (S, C, \alpha|_C)$ induced by C is *sensible*, i.e. that there exists at least one constructor ground term $t \in \mathrm{Term}_s(C)$ for each sort $s \in S$. The other function symbols $D \equiv F - C$ are called *defined*. Intuitively, constructor ground terms are evaluated to *defined data items* in an algebra; the other elements in the universe of the algebra represent undefined values.

Analogously, the set V of variables is divided into a set V^C of *constructor* variables and a set V^G of *general* variables. Intuitively, constructor variables may be evaluated only to data items whereas general variables may be evaluated to arbitrary values. This distinction also influences the notions for substitutions.

Definition 2.1 (Constructor / Inductive Substitutions) A substitution σ is called a *constructor substitution* if $\sigma(V^C) \subseteq \mathrm{Term}(C, V^C)$, i.e. if every constructor variable is bound to a constructor term. A substitution σ is called an *inductive substitution* if $\sigma(V^C) \subseteq \mathrm{Term}(C)$ and $\sigma(V^G) \subseteq \mathrm{Term}(F, V^G)$. □

In QuodLibet, formulas are represented as clauses with implicitly universally quantified variables. A *clause* is a disjunctively combined finite sequence of literals. A *literal* λ is an atom A or its negation $\neg A$. The conjugate $\overline{\lambda}$ of literal λ is defined by

$$\overline{\lambda} = \begin{cases} A & \text{if } \lambda \equiv \neg A \\ \neg\lambda & \text{otherwise} \end{cases}$$

Atoms are constructed from terms using three predefined predicate symbols:

- An *equality* atom $t = u$ consists of two terms of the same sort. These atoms enable the formulation of equality-based specifications.
- A *definedness* atom `def` t is used for expressing definedness properties of (partial) operators in the form of *domain lemmas*.
- An *order* atom $w_1 < w_2$ consists of two weights. It explicitly represents order obligations resulting from applications of induction hypotheses. A *weight* w is a finite sequence of terms $(u_1, \ldots, u_n)$ (with $n \leq k_0$ for a fixed $k_0 \in \mathbb{N}$).

  A concrete induction order is defined in two steps:

1. Weights $w_1 \equiv (u_1, \ldots, u_n)$ and $w_2 \equiv (v_1, \ldots, v_m)$ are compared w.r.t. a fixed wellfounded order. Basically, this order is the lexicographic order induced by the term lengths of the constructor terms that are equal to the terms in weights, i.e. $(|\hat{u}_1|, \ldots, |\hat{u}_n|) <_{\mathbb{N}}^{\text{lex}} (|\hat{v}_1|, \ldots, |\hat{v}_m|)$ where $\hat{u}_i$ and $\hat{v}_j$ are constructor terms equal to u_i and v_j, respectively, for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$.

   For this order to be welldefined, each term may at most be equal to one constructor term. Particularly, we require that the constructors are *free*: if, for two constructor terms $t_1^C, t_2^C \in \text{Term}(C, V^C)$, $t_1^C =_R t_2^C$ w.r.t. the considered specification $\texttt{spec} = (\Sigma, C, R)$ (cf. Definition 2.4), then $t_1^C \equiv t_2^C$.

2. To achieve flexibility for defining the induction order, weights are initially given by weight variables that may be instantiated to term tuples later on.

Usually, we represent clauses by upper Greek letters such as Γ and Δ; and literals by lower Greek letter such as λ. We write Γ, λ, Δ to represent a clause consisting of the literals in Γ, literal λ and the literals in Δ, successively. Abusing the notation, we also represent this in set notation—i.e. $\Gamma \cup \{\lambda\} \cup \Delta$—to improve readability in continuous text.[2] We extend the notions introduced in Section 2.1 to atoms, literals and clauses if suitable.

A *conditional* equation is an expression of the form $l = r \Leftarrow \Delta$ where Δ is a conjunctive sequence of *condition* literals. The conditional equation may be represented in clausal form as $\{l = r\} \cup \overline{\Delta}$ where $\overline{\Delta}$ is the disjunctive sequence of conjugated condition literals in Δ. A conditional equation may be applied to replace an instance $l\sigma$ of l by the same instance $r\sigma$ of r. Therefore, it is also called a (conditional) *rewrite rule*. To guarantee that the constructors of admissible specifications are free, we restrict the rewrite rules that can be used in specifications.

Definition 2.2 (Defining Rule) Let $\Sigma = (S, F, \alpha)$ be a signature and $C \subseteq F$ be a set of constructors for Σ. A rewrite rule $l = r \Leftarrow \Delta$ is called *defining rule* if

- $l \in \text{Term}(F, V) - \text{Term}(C, V)$ and
- Δ does not contain a literal of the form $\neg\texttt{def}\ t$.

□

Definition 2.3 (Specifications with Free Constructors)
A *specification* $\texttt{spec} = (\Sigma, C, R)$ with free constructors consists of a signature $\Sigma = (S, F, \alpha)$, a set of constructors $C \subseteq F$ such that the induced signature Σ^C is sensible and a set R of defining rules. □

The semantics of such a specification is given by the class of all data models of the specification (cf. Definition 2.9). For this semantics to be welldefined, the class of data models must not be empty. This is guaranteed for admissible specifications (cf. Definition 2.5) which essentially call for confluence of the rewrite relation $\longrightarrow_R$ induced by the defining rules R. To define this rewrite relation, we have to fix how the evaluation of the conditions is operationalized. In QuodLibet, this is done in a constructive way with constructor ground terms.

[2] Set notation is also used within the system itself. Therefore, the examples are represented in this way.

Definition 2.4 (Rewrite Relation $\longrightarrow_R$) Let $\texttt{spec} = (\Sigma, C, R)$ be a specification with free constructors. The *rewrite relation* $\longrightarrow_R$ induced by R is defined by $\longrightarrow_R = \bigcup_{i \in \mathbb{N}} \longrightarrow_i$ with

- $\longrightarrow_0 = \varnothing$ and
- $t_1 \longrightarrow_{i+1} t_2$ if there is a rewrite rule $l = r \Leftarrow \Delta$ in R, a position $p \in Pos(t_1)$ and an inductive substitution σ such that
  1. $t_1/p \equiv l\sigma$
  2. $t_2 \equiv t_1[r\sigma]_p$
  3. for each $u = v$ in Δ, $u\sigma \downarrow_i v\sigma$
  4. for each $\texttt{def}\ u$ in Δ, there is a term $\hat{u} \in \mathrm{Term}(C)$ such that $u\sigma \xrightarrow{*}_i \hat{u}$
  5. for each $u \neq v$ in Δ, there are terms $\hat{u}, \hat{v} \in \mathrm{Term}(C)$ such that $u\sigma \xrightarrow{*}_i \hat{u}$, $v\sigma \xrightarrow{*}_i \hat{v}$ and $\hat{u} \not\equiv \hat{v}$
  6. for each $(u_1, \ldots, u_n) < (v_1, \ldots, v_m)$ in Δ,
     - there are a $k \leq \min(n, m)$ and terms $\hat{u}_j, \hat{v}_j \in \mathrm{Term}(C)$ such that $u_j\sigma \xrightarrow{*}_i \hat{u}_j$, $v_j\sigma \xrightarrow{*}_i \hat{v}_j$ for $j \in \{1, \ldots, k\}$ and $(|\hat{u}_1|, \ldots, |\hat{u}_k|) <^{\mathrm{lex}}_{\mathbb{N}} (|\hat{v}_1|, \ldots, |\hat{v}_k|)$ or
     - $n < m$ and there are terms $\hat{u}_j, \hat{v}_j \in \mathrm{Term}(C)$ such that $u_j\sigma \xrightarrow{*}_i \hat{u}_j$, $v_j\sigma \xrightarrow{*}_i \hat{v}_j$ for $j \in \{1, \ldots, n\}$ and $(|\hat{u}_1|, \ldots, |\hat{u}_n|) = (|\hat{v}_1|, \ldots, |\hat{v}_n|)$

     and
  7. for each $\neg((u_1, \ldots, u_n) < (v_1, \ldots, v_m))$ in Δ,
     - there are a $k \leq \min(n, m)$ and terms $\hat{u}_j, \hat{v}_j \in \mathrm{Term}(C)$ such that $u_j\sigma \xrightarrow{*}_i \hat{u}_j$, $v_j\sigma \xrightarrow{*}_i \hat{v}_j$ for $j \in \{1, \ldots, k\}$ and $(|\hat{v}_1|, \ldots, |\hat{v}_k|) <^{\mathrm{lex}}_{\mathbb{N}} (|\hat{u}_1|, \ldots, |\hat{u}_k|)$ or
     - $m \leq n$ and there are terms $\hat{u}_j, \hat{v}_j \in \mathrm{Term}(C)$ such that $u_j\sigma \xrightarrow{*}_i \hat{u}_j$, $v_j\sigma \xrightarrow{*}_i \hat{v}_j$ for $j \in \{1, \ldots, m\}$ and $(|\hat{u}_1|, \ldots, |\hat{u}_m|) = (|\hat{v}_1|, \ldots, |\hat{v}_m|)$.

□

Definition 2.5 (Admissible Specifications with Free Constructors)
A specification $\texttt{spec} = (\Sigma, C, R)$ with free constructors as defined in Definition 2.3 is called *admissible* if

- $\longrightarrow_R$ is confluent and
- for each defining rule $l = r \Leftarrow \Delta$ and for each term $t \in \mathrm{Term}(F, V) - \mathrm{Term}(C, V^C)$ occurring (on top-level) in a negated equation or a negated order atom in Δ there is a literal $\texttt{def}\ t$ in Δ.

□

Since we want to model non-terminating operators, we cannot use confluence criteria that presuppose termination of the rewrite relation. Instead, QuodLibet employs an easily testable, sufficient confluence criterion based on complementary critical pairs.

Definition 2.6 (Complementary Critical Pairs) Let $\mathtt{spec} = (\Sigma, C, R)$ be a specification with free constructors.

(i) Let $l_i = r_i \Leftarrow \Delta_i$ be a rewrite rule in R with $X_i := V(l_i, r_i, \Delta_i)$ for $i \in \{0, 1\}$. Assume w.l.o.g. $X_0 \cap X_1 = \varnothing$. If there is a non-variable position $p \in Pos(l_1)$ such that $\sigma = mgu(l_0, l_1/p)$ exists and $l_1[r_0]_p\sigma \neq r_1\sigma$, then

$$(l_1[r_0]_p\sigma, \Delta_0\sigma),\ (r_1\sigma, \Delta_1\sigma)$$

is a (non-trivial) *critical pair*. The set of all critical pairs between rewrite rules in R is denoted by $\mathrm{CP}(R)$.

(ii) The above critical pair is *complementary* if

- there are $u, v \in \mathrm{Term}(F, V)$ and an $i \in \{0, 1\}$ such that $u \doteq v$[3] occurs in $\Delta_i\sigma$ and $u \neq v$ occurs in $\Delta_{1-i}\sigma$; or
- there are $t \in \mathrm{Term}(F, V)$ and $\hat{u}, \hat{v} \in \mathrm{Term}(C)$ with $\hat{u} \neq \hat{v}$ such that $t \doteq \hat{u}$ occurs in $\Delta_0\sigma$ and $t \doteq \hat{v}$ occurs in $\Delta_1\sigma$; or
- there are weights w_1, w_2 and an $i \in \{0, 1\}$ such that $w_1 < w_2$ occurs in $\Delta_i\sigma$ and $\neg(w_1 < w_2)$ occurs in $\Delta_{1-i}\sigma$.

$\square$

For the following confluence criterion two additional notions are required: A rewrite system R is *left-linear* if the left-hand side l of each rewrite rule $l = r \Leftarrow \Delta$ in R is linear, i.e. $|l|_x \leq 1$ for each $x \in V(l)$. R is *effectively quasi-normal* if for each rewrite rule $l = r \Leftarrow \Delta$ in R, and for each equality $t_1 = t_2$ in Δ, there is one t_i, $i \in \{1, 2\}$, such that $t_i \in \mathrm{Term}(C, V^C)$ or $\mathtt{def}\ t_i$ occurs in Δ.

Theorem 2.7 (Confluence Criterion for $\longrightarrow_R$) Let $\mathtt{spec} = (\Sigma, C, R)$ be a specification with free constructors such that R is left-linear and effectively quasi-normal. If each critical pair in $\mathrm{CP}(R)$ is complementary, then $\longrightarrow_R$ is confluent. $\square$

A proof of this theorem can be found in [Wir05a].

2.2.1.2 Semantics

Let $\mathtt{spec} = (\Sigma, C, R)$ be an admissible specification with free constructors, and $\mathcal{A} = (A, F^{\mathcal{A}})$ be a Σ-algebra. We refine the notion for valuations $\varphi : V \to A$ to account for the partitioning of variables into constructor and general variables: Let $A_s^C = \{t^{\mathcal{A}} \mid t \in \mathrm{Term}_s(C)\} \subseteq A_s$ be the set of defined data items of A_s. We consider only valuations φ with $\varphi(x) \in A_s^C$ for each $x \in V_s^C$.

To define the semantics of order atoms formally, we lift evaluations from terms to weights, i.e. tuples of terms: $\mathrm{eval}_\varphi^{\mathcal{A}}((t_1, \ldots, t_n)) = (\mathrm{eval}_\varphi^{\mathcal{A}}(t_1), \ldots, \mathrm{eval}_\varphi^{\mathcal{A}}(t_n))$. Furthermore, we use the length of constructor terms to define a semantical order $\leq_{\mathcal{A}}$ for each Σ-algebra:

[3] $u \doteq v$ stands for $u = v$ or $v = u$. Analogously, $u \dot{\neq} v$ stands for $u \neq v$ or $v \neq u$.

$a_1 \leq_{\mathcal{A}} a_2$ if

- $a_1 = a_2$ or
- there are $t_1, t_2 \in \mathrm{Term}(C)$ such that $t_1^{\mathcal{A}} = a_1$, $t_2^{\mathcal{A}} = a_2$ and $|t_1| < |t_2|$.

Definition 2.8 (Models) Let $\mathtt{spec} = (\Sigma, C, R)$ be an admissible specification with free constructors, and $\mathcal{A} = (A, F^{\mathcal{A}})$ be a Σ-algebra.

(i) Let $\varphi : V \to A$ be a valuation. Then $\mathcal{A}$ *with* φ *satisfies*

- an equation $t_1 = t_2$ if $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1) = \mathrm{eval}_{\varphi}^{\mathcal{A}}(t_2)$,
- a definedness atom $\mathtt{def}\ t$ for $t \in \mathrm{Term}_s(F, V)$ if $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t) \in A_s^C$ and
- an order atom $w_1 < w_2$ if $\mathrm{eval}_{\varphi}^{\mathcal{A}}(w_1) <_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_{\varphi}^{\mathcal{A}}(w_2)$.

$\mathcal{A}$ satisfies a negative literal $\neg\lambda$ with φ if $\mathcal{A}$ does not satisfy λ with φ. Finally, $\mathcal{A}$ satisfies a clause Γ with φ if there is at least one literal in Γ that $\mathcal{A}$ satisfies with φ.

(ii) A clause Γ is *valid* in $\mathcal{A}$ if $\mathcal{A}$ satisfies Γ with every valuation $\varphi : V \to A$. This is denoted by $\mathcal{A} \models \Gamma$. Let K be a class of Σ-algebras and E be a set of clauses. We write $K \models E$ iff $\mathcal{A} \models \Gamma$ for every $\mathcal{A} \in K$ and for every Γ in E.

(iii) $\mathcal{A}$ is a Σ-*model* of $\mathtt{spec}$ if each defining rule in R is valid in $\mathcal{A}$. The class of all Σ-models of $\mathtt{spec}$ is denoted by $\mathrm{Mod}(\mathtt{spec})$.

□

Instead of all models we are interested only in those models that do not equalize any constructor ground terms.

Definition 2.9 (Data Models / Standard Data Model $\mathcal{M}(\mathtt{spec})$)
Let $\mathtt{spec} = (\Sigma, C, R)$ be an admissible specification with free constructors. A Σ-model $\mathcal{A}$ of $\mathtt{spec}$ is a *data model* of $\mathtt{spec}$ if, for all different constructor ground terms $t_1, t_2 \in \mathrm{Term}(C)$, $t_1^{\mathcal{A}} \neq t_2^{\mathcal{A}}$. Let $\mathrm{DMod}(\mathtt{spec})$ denote the class of all data models of $\mathtt{spec}$. The *standard data model* $\mathcal{M}(\mathtt{spec})$ is defined as the quotient algebra $\mathrm{Term}(F, V^G)/\overset{*}{\longleftrightarrow}_R$. □

In general, the class of data models for an arbitrary specification may be empty. But for admissible specifications, the following theorem is proved in [KW97]:

Theorem 2.10 (Existence of Standard Data Model) Let $\mathtt{spec} = (\Sigma, C, R)$ be an admissible specification with free constructors. Then the standard data model $\mathcal{M}(\mathtt{spec})$ is a data model of $\mathtt{spec}$. □

Thus, using the class of all data models to define inductive validity results in a welldefined semantics.

Definition 2.11 (Inductive Validity)
Let $\mathtt{spec} = (\Sigma, C, R)$ be an admissible specification with free constructors. A clause Γ is *inductively valid* in $\mathtt{spec}$ if $\mathrm{DMod}(\mathtt{spec}) \models \Gamma$. □

Using the class of all data models to define the semantics instead of just one single model such as the standard data model provides the advantage that the semantics is *monotonic* w.r.t. (constructor-consistent) extensions of a specification: In short, inductively valid lemmas remain inductively valid in extended specifications.

Definition 2.12 (Constructor-Consistent Extension of a Specification)
For $i \in \{0, 1\}$ let $\mathtt{spec}_i = (\Sigma_i, C_i, R_i)$ be an admissible specification with free constructors where $\Sigma_i = (S_i, F_i, \alpha_i)$, and let $V_i = (V_{i,s})_{s \in S_i}$ be a variable system for Σ_i. We say that $\mathtt{spec}_1$ is a *constructor-consistent extension* of $\mathtt{spec}_0$ if the following conditions are met:

1. $S_0 \subseteq S_1$, $F_0 \subseteq F_1$, $C_0 \subseteq C_1$ and $\alpha_0 \subseteq \alpha_1$

2. $V_{0,s} = V_{1,s}$ for each $s \in S_0$

3. $R_0 \subseteq R_1$ and

4. for each $c \in C_1 - C_0$ with $\alpha_1(c) = s_1 \ldots s_n s$ we have $s \notin S_0$.

□

Theorem 2.13 (Monotonicity of Constructor-Consistent Extensions)
Let $\mathtt{spec}_1$ be a constructor-consistent extension of $\mathtt{spec}_0$, and Γ be a clause (over Σ_0 and V_0). If $\mathrm{DMod}(\mathtt{spec}_0) \models \Gamma$, then $\mathrm{DMod}(\mathtt{spec}_1) \models \Gamma$. □

Again, this theorem is proved in [KW97].

2.2.2 Inference Rules

In QUODLIBET, inductive proofs are performed with an inference system working on goals. A *goal* $\langle \Gamma; w \rangle$ consists of a clause Γ to be proved and a weight w for the definition of the induction order. The inference system is based on a *sequent calculus*. The inference rules are applied *reductively*, i.e. a goal (*conclusion*) is reduced to a (possibly empty) sequence of new subgoals (*premises*). Therefore, we perform goal-oriented backward-reasoning.

Naturally, the inference rules are important for the automation of the proof process presented in this thesis. Therefore, we list the inference rules with all their technical details in this section. Additionally, we supplement the formal description with informal remarks about their typical usage to ease comprehension. A complete example proof is presented in Section 2.2.3. Further motivation and examples can be found in [Küh00].

We precede the presentation of the inference rules with three preliminary sections: In Section 2.2.2.1, we define local properties that all the inference rules possess. These local properties are sufficient to ensure global properties of whole proofs represented by proof state graphs. This is described in Section 2.2.2.2. In Section 2.2.2.3, we summarize additional notions that are required for the formal presentation of the inference rules.

2.2.2.1 Local Properties of Inference Rules

Inference rules in QuodLibet are parameterized schemes of the form

<rule name> <parameters>

$$\frac{\langle\Gamma; w\rangle}{\langle\Gamma_1; w_1\rangle \ldots \langle\Gamma_n; w_n\rangle}$$

with $\langle\Pi_1; \hat{w}_1\rangle^{U_1} \ldots \langle\Pi_k; \hat{w}_k\rangle^{U_k}$

if <applicability conditions>

where $n, k \in \mathbb{N}$ and $U_i \in \{\mathcal{L}, \mathcal{I}\}$ for $i \in \{1, \ldots, k\}$:

<rule name> is the name of the inference rule.

<parameters> are the meta-variables that have to be instantiated to get a concrete instance.

$\langle\Gamma; w\rangle$ stands for the goal that is reduced by the inference rule.

$\langle\Gamma_1; w_1\rangle \ldots \langle\Gamma_n; w_n\rangle$ represent the new subgoals resulting from the application of the inference rule.

$\langle\Pi_1; \hat{w}_1\rangle^{U_1} \ldots \langle\Pi_k; \hat{w}_k\rangle^{U_k}$ are the axioms or lemmas that are applied by the inference rule. If $U_i = \mathcal{I}$, then $\langle\Pi_i; \hat{w}_i\rangle$ is applied as induction hypothesis; if $U_i = \mathcal{L}$, then $\langle\Pi_i; \hat{w}_i\rangle$ is applied non-inductively.

<applicability conditions> are the conditions that have to be fulfilled by the goal the inference rule is applied to.

An inference rule is called *applicative* if $k > 0$; otherwise it is called *non-applicative*.

The inference rules make use of semantical properties of the predefined predicate symbols and the clause form of formulas. In comparison to non-inductive inference systems, the formulation of local soundness criteria for inference systems that may apply lemmas as induction hypothesis with explicit order constraints is technically more involved. The definition depends on counterexamples: If a clause Γ is not inductively valid in a data model $\mathcal{A}$, then there exists a counterexample. Roughly speaking, applications of sound inference rules make counterexamples "smaller". Therefore, the local soundness properties of the inference rules guarantee that there does not exist any $\mathcal{A}$-counterexample for any data model $\mathcal{A}$ if a proof within the inference system is found. In QuodLibet, *$\mathcal{A}$-counterexamples* (for $\langle\Gamma; w\rangle$) have the form $(\langle\Gamma; w\rangle, \sigma, \varphi)$ where σ is an inductive substitution and φ is a valuation for V^G such that $\mathcal{A}$ does not satisfy $\Gamma\sigma$ with φ.

Definition 2.14 (Soundness of Inference Rules) We call an inference rule *sound* if for any admissible specification `spec` with free constructors, for any instance

$$\frac{\langle\Gamma; w\rangle}{\langle\Gamma_1; w_1\rangle \ \ldots\ \langle\Gamma_n; w_n\rangle} \quad \text{with} \quad \langle\Pi_1; \hat{w}_1\rangle^{U_1}, \ldots, \langle\Pi_k; \hat{w}_k\rangle^{U_k}$$

of the given inference rule, for any $\mathcal{A} \in \mathrm{DMod}(\texttt{spec})$, and for any $\mathcal{A}$-counterexample of the form $(\langle\Gamma; w\rangle, \sigma, \varphi)$, one of the following statements holds:

(1) There is an $i \in \{1, \ldots, n\}$ and an $\mathcal{A}$-counterexample of the form $(\langle \Gamma_i; w_i \rangle, \tau, \psi)$ such that $\mathrm{eval}^{\mathcal{A}}_{\psi}(w_i\tau) \leq^{\mathrm{lex}}_{\mathcal{A}} \mathrm{eval}^{\mathcal{A}}_{\varphi}(w\sigma)$.

(2) There is a $j \in \{1, \ldots, k\}$ such that $U_j = \mathcal{L}$ and Π_j is not inductively valid w.r.t. `spec`.

(3) There is a $j \in \{1, \ldots, k\}$ and an $\mathcal{A}$-counterexample of the form $(\langle \Pi_j; \hat{w}_j \rangle, \tau, \psi)$ such that $U_j = \mathcal{I}$ and $\mathrm{eval}^{\mathcal{A}}_{\psi}(\hat{w}_j\tau) <^{\mathrm{lex}}_{\mathcal{A}} \mathrm{eval}^{\mathcal{A}}_{\varphi}(w\sigma)$.

□

Additionally, the inference rules of QuodLibet do not introduce any new counterexample.

Definition 2.15 (Safeness of Inference Rules) An inference rule is called *safe* if, for any admissible specification `spec` with free constructors, and for any instance

$$\frac{\langle \Gamma; w \rangle}{\langle \Gamma_1; w_1 \rangle \ \ldots \ \langle \Gamma_n; w_n \rangle} \quad \text{with} \quad \langle \Pi_1; \hat{w}_1 \rangle^{U_1}, \ldots, \langle \Pi_k; \hat{w}_k \rangle^{U_k}$$

of the given inference rule, inductive validity of each of the clauses in $\{\Gamma, \Pi_1, \ldots, \Pi_k\}$ w.r.t. `spec` implies inductive validity of each of the clauses in $\{\Gamma_1, \ldots, \Gamma_n\}$ w.r.t. `spec`. □

Theorem 2.16 (Soundness and Safeness Property for QuodLibet)
All the inference rules presented in this section are sound and safe for admissible specifications with free constructors. □

This theorem is proved in [Küh00]. In most cases, the safeness of the inference rules is ensured because the clauses of the new subgoals contain the clause of the original goal. Therefore, we do not provide any further explanations about the safeness of the inference rules but we only argue about their soundness in an informal way.

2.2.2.2 Proof State Graphs

Proof state graphs serve two purposes: On the one hand, they provide means for representing proof attempts of lemmas to the user so that he can call a tactic for a goal to start an automatic proof attempt, or apply an inference rule to a goal manually. On the other hand, they are used for managing the dependencies that result from the applications of inference rules. Thus, we can pose easy requirements on proof state graphs that guarantee the inductive validity of the corresponding lemmas due to the local soundness properties of the inference rules. We introduce the necessary concepts and notions for proof state graphs on an informal level only. For a formal treatment we refer to [Küh00].

In simple terms, a consists of one proof state tree for each lemma to be proved. A *proof state tree* is a labeled directed bipartite graph consisting of *goal* and *inference nodes*. The *root* goal node of a proof state tree is labeled with a goal containing the lemma to be proved. An inference node is labeled with the inference rule applied to its parent which is a goal node. Its n children ($n \geq 0$) are again goal nodes and represent the new subgoals created by the inference rule. In the simplest case, inference rules are applied only to those goal nodes that are leaves—resulting in a single *proof attempt*. Furthermore, it is also possible to start several proof attempts in parallel. To do so, a goal node may have several inference nodes as children. Any of these subtrees represents a different proof attempt.

A goal node is *open* if it does not possess any children, i.e. if no inference rule has been applied to it. A proof attempt is *closed* if there are no open goal nodes in the proof attempt, i.e. if all its leaves are inference nodes. Note that a closed proof attempt is not sufficient for inductive validity—e.g. the inductive validity of the lemma labeling the root goal node—as QuodLibet allows for the application of yet unproved lemmas. Therefore, a proof state graph additionally contains dependencies for the applied lemmas. A proof attempt is a *proof* if it is closed and all non-inductively applied lemmas are themselves inductively valid. From the local properties of the inference rules we get the following:

Soundness: Due to the local soundness property, all the formulas that label goal nodes of a proof are inductively valid. In particular, this holds true for the lemma labeling a proved root goal.

Safeness: Due to the local safeness property, if a proof attempt contains a goal labeled with a formula that is not inductively valid such as the empty clause, then the lemma labeling the root goal or one of the lemmas applied non-inductively within the proof attempt is not inductively valid.

A proof state tree may be interpreted as and/or-tree with goal nodes as or-nodes and inference nodes as and-nodes: Only one proof attempt needs to be successful.

2.2.2.3 Additional Notions for Inference Rules

We summarize additional notions required for the formal definition of the inference rules. We use the following meta-variables without further explanation:

- m, n, j, k for natural numbers,
- d for a depth, i.e. a natural number or ω,[4]
- t, u, v, l, r for terms,
- p for positions of terms in literals or terms,
- w for weights,
- λ for literals and
- Γ, Π, Λ for clauses.

With $\Gamma[m]$ we denote the mth literal in clause Γ. We extend the notions for replacements to clauses: With $\Gamma[t]_{m.p}$ we denote the clause that is derived from Γ by replacing the subterm at position p in the mth literal of Γ with term t. With $\Gamma[\lambda]_m$ we denote the clause that is derived from Γ by replacing the mth literal of Γ with λ. With $\Gamma[-]_m$ the mth literal is eliminated from Γ.

[4] ω is represented by -1 in QuodLibet.

Often we do not want to distinguish between (negated) equality atoms with interchanged left-hand and right-hand side. We denote this type of equality between literals with $=_{\mathrm{lit}}$. Formally, $\lambda =_{\mathrm{lit}} \lambda'$ if

- $\lambda \equiv \lambda'$ or
- $\lambda \equiv (u = v)$ and $\lambda' \equiv (v = u)$ or
- $\lambda \equiv (u \neq v)$ and $\lambda' \equiv (v \neq u)$.

A literal λ *occurs* in a clause Γ if there is a literal λ' in Γ with $\lambda =_{\mathrm{lit}} \lambda'$. A clause Γ *contains* a clause Π if each literal in Π occurs in Γ.

We define the *minimal difference positions* $\mathrm{MinDifPos}(t_1, t_2, d)$ where the terms t_1 and t_2 differ w.r.t. a maximal depth d. Formally, $\mathrm{MinDifPos}(t_1, t_2, d)$ are the minimal positions in $\{\mathrm{prefix}(p, d) \mid p \in \mathit{Pos}(t_1) \cap \mathit{Pos}(t_2) \text{ with } \mathrm{top}(t_1/p) \not\equiv \mathrm{top}(t_2/p)\}$ (cf. Section 2.1 for the definitions of prefix and top).

Analogously, we define the *minimal non-constructor positions* $\mathrm{MinNonCPos}(t)$ of a term t as the minimal positions in $\{p \in \mathit{Pos}(t) \mid \mathrm{top}(t/p) \notin (C \cup V^C)\}$.

The *C-front* of a term t is a constructor term $t^C \in \mathrm{Term}(C, V^C)$ that is generated from t by consistently replacing the terms at minimal non-constructor positions with new constructor variables of the same sort. Consistently means that for each $p_1, p_2 \in \mathrm{MinNonCPos}(t)$, $t/p_1 \equiv t/p_2$ iff $t^C/p_1 \equiv t^C/p_2$.

A set $\{\sigma_1, \ldots, \sigma_n\}$ of constructor substitutions is a *cover set of substitutions* (for a goal $\langle \Gamma; w \rangle$) if for every inductive substitution σ there is a $j \in \{1, \ldots, n\}$ and an inductive substitution τ such that $x\sigma \equiv x\sigma_j\tau$ (for each $x \in V(\Gamma, w)$).

The *case analysis resulting from literals* $\lambda_1, \ldots, \lambda_n$ (for $n > 0$) consists of the clauses $\Lambda_1, \ldots, \Lambda_n, \Lambda$ such that

- $\Lambda_i \equiv \overline{\lambda_i}, \lambda_{i-1}, \ldots, \lambda_1$ for $i \in \{1, \ldots, n\}$ and
- $\Lambda \equiv \lambda_n, \ldots, \lambda_1$.

The set of *definedness conditions* $\mathrm{DefCond}(\mu, \Gamma)$ of a substitution μ and a clause Γ is defined as $\mathrm{DefCond}(\mu, \Gamma) = \{\neg\mathtt{def}\ x\mu \mid x \in V^C \cap V(\Gamma) \text{ and } x\mu \notin \mathrm{Term}(C, V^C)\}$.

2.2.2.4 Inference Rules for Simple Tautologies

With the first three inference rules we can prove simple tautologies (cf. Figure 2.1). Therefore, the inference rules do not create any new subgoals.

`compl-lit` is applicable if the considered literals are complementary, i.e. one literal is the conjugate of the other. This inference rule is sound for all Σ-algebras due to the clausal form of the formulas.

`compl-lit` $m\ n$

$$\frac{\langle \Gamma; w \rangle}{}$$

if • $\Gamma[m] =_{\text{lit}} \overline{\Gamma[n]}$.

$\neq$`-taut` m

$$\frac{\langle \Gamma; w \rangle}{}$$

if • $\Gamma[m] = (t_1 \neq t_2)$
- $t_1, t_2 \in \text{Term}(C, V^C)$
- t_1 and t_2 are not unifiable.

$<$`-taut` m

$$\frac{\langle \Gamma; w \rangle}{}$$

if • $\Gamma[m] = (() < (u_1, \ldots, u_k))$
- $k > 0$.

Figure 2.1: Inference Rules for Simple Tautologies

$\neq$`-taut` is applicable if the considered literal is a negated equation with constructor terms at each side that are not unifiable. This inference rule is sound as we consider data models for specifications with free constructors only. Therefore, different constructor ground terms are always mapped to different data items.

$<$`-taut` is applicable if the considered literal is an order atom with an empty tuple as left-hand side but with a non-empty tuple as right-hand side. This inference rule is sound due to the properties of the lexicographic order.

2.2.2.5 Inference Rules for Decomposing Atoms

The inference rules for decomposing atoms (cf. Figure 2.2) may be used for proving simple tautologies. In this case, they do not create any new subgoals. But they may also be used for generating new subgoals that contain one simpler literal that is derived from the considered literal.

$=$`-decomp` is applicable if the considered literal is an equation such that the top-level symbols of both sides are identical. To prove the equality $t_1 = t_2$, it suffices to prove $t_1/p = t_2/p$ for each minimal difference position p w.r.t. a maximal depth d. For a new equality whose complement is already present in the original clause we do not have to create a new subgoal since it can be proved with inference rule `compl-lit` immediately. Therefore, these subgoals can be cut off. This results in the identification of cut-off literals in the original goal as described in Chapter 5.

$=$-decomp m d

$$\frac{\langle \Gamma; w\rangle}{\langle u_1 = v_1, \Gamma; w\rangle \ \ldots \ \langle u_k = v_k, \Gamma; w\rangle}$$

if
- $\Gamma[m] = (t_1 = t_2)$
- $d > 0$
- $\mathrm{top}(t_1) = \mathrm{top}(t_2)$
- $u_1 = v_1, \ldots, u_k = v_k$ are exactly those equations in

  $$\{ t_1/p \doteq t_2/p \mid p \in \mathrm{MinDifPos}(t_1, t_2, d) \}$$

  whose complements do not occur in $\Gamma[-]_m$.

def-decomp m

$$\frac{\langle \Gamma; w\rangle}{\langle \texttt{def}\ u_1, \Gamma; w\rangle \ \ldots \ \langle \texttt{def}\ u_k, \Gamma; w\rangle}$$

if
- $\Gamma[m] = \texttt{def}\ t$
- $\mathrm{top}(t) \in (C \cup V^C)$
- $\texttt{def}\ u_1, \ldots, \texttt{def}\ u_k$ are exactly those definedness atoms in

  $$\{ \texttt{def}\ t/p \mid p \in \mathrm{MinNonCPos}(t) \}$$

  whose complements do not occur in $\Gamma[-]_m$.

$<$-decomp m

$$\frac{\langle \Gamma; w\rangle}{\langle \texttt{def}\ u_1, \Gamma; w\rangle \ \ldots \ \langle \texttt{def}\ u_k, \Gamma; w\rangle}$$

if
- $\Gamma[m] = (t_1 < t_2)$
- $\mathrm{top}(t_2) \in C$
- there are $\hat{t}_1, \hat{t}_2 \in \mathrm{Term}(C, V^C)$ such that
  - for $i \in \{1, 2\}$, $\hat{t}_i$ is a C-front for t_i
  - for each $p_1 \in \mathrm{MinNonCPos}(t_1)$ and $p_2 \in \mathrm{MinNonCPos}(t_2)$, $t_1/p_1 = t_2/p_2$ iff $\hat{t}_1/p_1 = \hat{t}_2/p_2$
  - $|\hat{t}_1| < |\hat{t}_2|$ and $|\hat{t}_1|_x \leq |\hat{t}_2|_x$ for every $x \in V^C$
  - $\texttt{def}\ u_1, \ldots, \texttt{def}\ u_k$ are exactly those definedness atoms in

    $$\{ \texttt{def}\ t_i/p \mid i \in \{1, 2\} \wedge p \in \mathrm{MinNonCPos}(t_i) \}$$

    whose complements do not occur in $\Gamma[-]_m$.

Figure 2.2: Inference Rules for Decomposing Atoms

If t_1 and t_2 are identical, no new subgoals are created. This is the typical usage of the inference rule.

`def-decomp` is applicable if the considered literal is a definedness atom that contains a constructor symbol at top-level. To prove `def` t, it is sufficient to prove `def` t/p for each minimal non-constructor position p in t. Again, we cut off those subgoals that can be proved with inference rule `compl-lit` immediately, identifying cut-off literals in the original clause.

If t is a constructor term, no new subgoals are created.

$<$`-decomp` is applicable if the considered literal is an order atom with one term at each side such that the order constraint is fulfilled for all data models provided that the terms are defined. The order constraint is checked using the C-fronts of both terms: If the length of the left-hand side is smaller than the length of the right-hand side and each variable occurs at most as often in the left-hand side as in the right-hand side of the C-fronts than the order constraint is fulfilled for defined terms. The definedness properties of the terms are checked as for inference rule `def-decomp` including the usage of cut-off literals.

If both terms are constructor terms, no new subgoals are created.

2.2.2.6 Inference Rules for Removing Redundant Literals

The inference rules presented in Figure 2.3 remove redundant literals. They “complement” the “tautological” inference rules presented in Figures 2.1 and 2.2: In short, if we can prove a clause valid because of a single literal then the conjugate of this literal cannot provide any additional information. Therefore, it can be removed from the clause without losing any information. The inference rules create exactly one new subgoal eliminating the redundant literal from the original clause. Theoretically, these inference rules do not have to be applied at all as they do not contain any new information. Their applications do not contribute to a proof in terms of Chapter 7. Nevertheless, their usage is indispensable in practice—at least when goals are presented to the user. They clean up the goals making the presentation more concise. Furthermore, they restrict proof search.

`mult-lit` complements inference rule `compl-lit`: It is applicable if the considered literals are equal. As two equal literals do not contain different information, one of the occurrences can be removed safely.

$=$`-removal` complements inference rule $\neq$`-taut`: It is applicable if the considered literal is an equation with constructor terms at each side that are not unifiable. In this case, the equation can never be fulfilled. Therefore, it can be removed from the clause safely.

$<$`-removal` complements inference rule $<$`-taut`: It is applicable if the considered literal is an order literal with an empty tuple as right-hand side. The literal can be removed safely because of the definition of the lexicographic order.

$\neq$`-removal` complements inference rule $=$`-decomp`: It is applicable if the equality of terms t_1 and t_2 of the considered negated equation $t_1 \neq t_2$ follows from the negation of the `mult-lit` m n

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_n; w\rangle}$$

if • $m \neq n$
• $\Gamma[m] =_{\text{lit}} \Gamma[n]$.

$=$`-removal` m

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_m; w\rangle}$$

if • $\Gamma[m] = (t_1 = t_2)$
• $t_1, t_2 \in \mathrm{Term}(C, V^C)$
• t_1 and t_2 are not unifiable.

$<$`-removal` m

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_m; w\rangle}$$

if • $\Gamma[m] = ((u_1, \ldots, u_k) < ())$.

$\neq$`-removal` m

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_m; w\rangle}$$

if • $\Gamma[m] = (t_1 \neq t_2)$
• $\mathrm{top}(t_1) = \mathrm{top}(t_2)$
• each atom in $\{t_1/p \neq t_2/p \mid p \in \mathrm{MinDifPos}(t_1, t_2, \omega)\}$ occurs in $\Gamma[-]_m$.

$\neg$`def-removal` m

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_m; w\rangle}$$

if • $\Gamma[m] = \neg\texttt{def}\ t$
• $\mathrm{top}(t) \in (C \cup V^C)$
• each atom in $\{\neg\texttt{def}\ t/p \mid p \in \mathrm{MinNonCPos}(t)\}$ occurs in $\Gamma[-]_m$.

Figure 2.3: Inference Rules for Removing Redundant Literals

`const-rewrite` $m\ n\ p$

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[t_2]_{m.p}; w\rangle}$$

if
- $m \neq n$
- $\Gamma[n] = (t_1 \dot{\neq} t_2)$
- $p \in Pos(\Gamma[m])$ and $\Gamma[m]/p = t_1$.

$\neq$`-unif` m

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_m\tau; w\tau\rangle}$$

if
- $\Gamma[m] = (t_1 \neq t_2)$
- $t_1, t_2 \in \mathrm{Term}(C, V^C)$
- $\tau = mgu(t_1, t_2)$ exists.

`ctr-var-add` $m\ x$

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[x \neq t]_m; w\rangle}$$

if
- $\Gamma[m] = \neg\mathtt{def}\ t$
- there is a sort s with $t \in \mathrm{Term}_s(F, V)$ and $x \in V_s^C - V(\Gamma, w)$.

Figure 2.4: Inference Rules for Making Use of Negative Literals

other literals in the clause, i.e. if the clause contains $t_1/p \neq t_2/p$ for each minimal difference position p of t_1 and t_2. Then, the considered literal does not contain any additional information and can be removed from the clause safely.

$\neg$`def-removal` complements inference rule `def-decomp`: It is applicable if the definedness property for term t of the considered negated definedness atom $\neg\mathtt{def}\ t$ follows from the negation of the other literals in the clause, i.e. if the clause contains $\neg\mathtt{def}\ t/p$ for each minimal non-constructor position p of t. Then, the considered literal does not contain any additional information and can be removed from the clause safely.

2.2.2.7 Inference Rules for Making Use of Negative Literals

A clause $\{\lambda_1, \ldots, \lambda_n\}$ may be interpreted as implication $\overline{\lambda_1} \rightarrow \lambda_2 \vee \cdots \vee \lambda_n$. Therefore, we may prove one the literals $\lambda_2, \ldots, \lambda_n$ assuming the negation of λ_1. The following inference rules make use of such negated literals (cf. Figure 2.4).

`const-rewrite` is applicable if the nth literal in the clause is a negated equation that contains a subterm of the mth literal (at position p). This subterm is replaced with the other side of the negated equation assuming its equality.

$\neq$-`unif` is applicable if the considered literal is a negated equation consisting of constructor terms that are unifiable. Assuming its negation it suffices to prove one of the remaining literals in the clause if the two terms are equal. Therefore, we can apply the most general unifier of the two terms to the other literals. Additionally, we can remove the negated equation from the clause as we have extracted all its information with the unifier.

Note that the weight is also instantiated with the unifier as we, henceforth, consider a special instance of the original goal.

`ctr-var-add` is applicable if the considered literal is a negated definedness atom $\neg$`def` t. Assuming its negation it suffices to prove one the remaining literals if t is defined, i.e. if it is equal to an arbitrary but fixed constructor term. This constructor term can be represented by a new constructor variable x (of the same sort) which does not occur in the original clause. Therefore, we may prove the remaining literals under the assumption that x is equal to t. Thus, we may replace $\neg$`def` t with $x \neq t$ in the new subgoal.

2.2.2.8 Further Inference Rules for Order Atoms

The fixed wellfounded lexicographic order used for evaluating order atoms in QuodLibet enjoys additional properties that are exploited by the inference rules depicted in Figure 2.5.

`tuple-<-reduct` is applicable if the considered literal is an order atom such that both tuples contain at least one element. As the order is a lexicographic one, the order constraint can be proved valid if this holds true for the first components of the tuples. Thus, the inference rule adds one order atom to the new subgoal in which the first components of the tuples are compared.

`tuple-=-reduct` is applicable if the considered literal is an order atom such that both tuples contain the same term as first element. In this case, the comparison w.r.t. the lexicographic order is decided by the remaining elements in the tuples. Thus, the inference rule replaces the old order atom in the original goal with an order atom without the first element in both tuples.

`<-mono` is applicable if the considered literal is an order atom such that each tuple consists of one term. Furthermore, p_1 and p_2 are positions in the terms of the left-hand and right-hand side, respectively, such that the remaining symbols in the contexts are constructor symbols and that the context on the left-hand side adds at most as much to the constructor length as the context on the right-hand side. Thus, if we can prove valid the order constraint resulting from the subterms at positions p_1 and p_2, then the original constraint is also valid. Therefore, the inference rule adds this order atom to the new subgoal.

`<-trans` is applicable if the considered literal is an order atom $w_1 < w_3$. The soundness of this inference rule depends only on the transitivity of the order relation. If we can prove the validity of $w_1 < w_2$ and $w_2 < w_3$ for an additional weight w_2, then the original order atom is also valid. Thus, the inference rule creates two new subgoals with one of the additional order constraints added to each subgoal.

`tuple-<-reduct` m

$$\frac{\langle \Gamma; w \rangle}{\langle t_1 < t_2, \Gamma; w \rangle}$$

if • $\Gamma[m] = ((t_1, u_1, \ldots, u_j) < (t_2, v_1, \ldots, v_k))$
• $j > 0 \ \vee \ k > 0$.

`tuple-=-reduct` m

$$\frac{\langle \Gamma; w \rangle}{\langle \Gamma[(u_1, \ldots, u_j) < (v_1, \ldots, v_k)]_m; w \rangle}$$

if • $\Gamma[m] = ((t, u_1, \ldots, u_j) < (t, v_1, \ldots, v_k))$.

`<-mono` $m\ p_1\ p_2$

$$\frac{\langle \Gamma; w \rangle}{\langle u_1 < u_2, \Gamma; w \rangle}$$

if • $\Gamma[m] = (t_1 < t_2)$
• for $i \in \{1, 2\}$, $p_i \in Pos(t_i) - \{\varepsilon\}$ and $t_i/p_i = u_i$
• there are $\hat{t}_1, \hat{t}_2 \in \mathrm{Term}(C, V^C)$ and $x_1, x_2 \in V^C - V(t_1, t_2)$ such that
  – for $i \in \{1, 2\}$, $\hat{t}_i = t_i[x_i]_{p_i}$
  – $|\hat{t}_1| \leq |\hat{t}_2|$
  – for every $x \in V^C - \{x_1, x_2\}$, $|\hat{t}_1|_x \leq |\hat{t}_2|_x$.

`<-trans` $m\ w_2$

$$\frac{\langle \Gamma; w \rangle}{\langle w_1 < w_2, \Gamma; w \rangle \ \langle w_2 < w_3, \Gamma; w \rangle}$$

if • $\Gamma[m] = (w_1 < w_3)$.

Figure 2.5: Further Inference Rules for Order Atoms

2.2.2.9 Inference Rules for Performing Case Splits

The inference rules in Figure 2.6 can be used for performing case splits.

`subst-add` is applicable if the given substitutions form a cover set of substitutions that bind only variables of the goal. The original goal—i.e. the clause and the weight—is instantiated with the cover set of substitutions creating one new subgoal for each substitution. The soundness of the inference rule is guaranteed because a cover set of substitutions defines a complete case split.

This inference rule is used, e.g., for the initial case split of inductive proofs that are based on constructor recursion.

`subst-add` $\sigma_1 \ldots \sigma_n$.

$$\frac{\langle \Gamma; w \rangle}{\langle \Gamma\sigma_1; w\sigma_1 \rangle \ \ldots \ \langle \Gamma\sigma_n; w\sigma_n \rangle}$$

if • $\{\sigma_1, \ldots, \sigma_n\}$ is a cover set of substitutions for $\langle \Gamma; w \rangle$.

`lit-add` $\lambda_1 \ldots \lambda_n$.

$$\frac{\langle \Gamma; w \rangle}{\langle \Lambda_1, \Gamma; w \rangle \ \ldots \ \langle \Lambda_n, \Gamma; w \rangle \ \langle \Lambda, \Gamma; w \rangle}$$

if • $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from literals $\lambda_1, \ldots, \lambda_n$ for $n > 0$.

Figure 2.6: Inference Rules for Performing Case Splits

`lit-add` is always applicable. It performs a cut, i.e. a case split w.r.t. a given list of literals $\lambda_1, \ldots, \lambda_n$. For each literal, one new subgoal is created where the original clause is handled provided that the literal is true. To be more precise, the literals of previous cases are downfolded to the right, i.e. the following cases assume that the previous literals are false. Additionally, one new subgoal is created that handles the case where all the literals are false. Therefore, we get a complete case split, and the inference rule is sound.

This inference rule is used, e.g., for the initial case split of inductive proofs that are based on destructor recursion.

2.2.2.10 Non-Inductive Applicative Inference Rules

The first two non-inductive applicative inference rules apply (conditional) lemmas for rewriting and subsumption, respectively (cf. Figure 2.7). As described in Section 2.2.1.1, a clause $\{\lambda_1, \ldots, \lambda_n\}$ can be interpreted as an implication $\overline{\lambda_1} \wedge \cdots \wedge \overline{\lambda_{n-1}} \Rightarrow \lambda_n$. A lemma is called *conditional* if $n > 1$. As in [Zha95], we fix one literal in the lemma clause by calling it the *head literal*; the conjugates of the other literals are called *condition literals*. For each inference step, we also fix one literal in the goal clause, called *focus literal*; the conjugates of the other literals are called *context literals*.

`lemma-rewrite` is applicable if the head literal—the nth literal—of the lemma clause is an equation $l \doteq r$ such that, for a substitution μ, $l\mu$ is equal to the subterm of the focus literal—the mth literal—of the goal clause at position p. The subterm is then replaced with $r\mu$ resulting in a *rewrite subgoal*. Furthermore, the instantiated condition literals have to be fulfilled in the "context": For each instantiated condition literal, a *condition subgoal* is created that essentially extends the original goal by the instantiated condition literal. Furthermore, for each constructor variable that is bound to a non-constructor term by substitution μ, a *definedness subgoal* is created that essentially extends the original goal by a definedness atom for the bound term. If an instantiated condition literal or definedness atom is equal to a context literal we say that it is *directly fulfilled* in the goal and the context literal is called a *cut-off* `lemma-rewrite` $m\ p\ \textit{PS-Tree}\ n\ \mu$

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle\ \ldots\ \langle \Lambda_n, \Gamma; w\rangle\ \langle \Lambda, \Gamma[r\mu]_{m.p}; w\rangle}$$

with $\langle \Pi; \hat{w}\rangle^{\mathcal{L}}$

if there is a position $p \in Pos(\Gamma[m])$, a substitution μ and a minimal clause Θ such that

- the goal labeling the root of the proof state tree *PS-Tree* is $\langle \Pi; \hat{w}\rangle$
- $\Pi[n] = (l \doteq r)$
- $\Gamma[m]/p = l\mu$
- $\Gamma[l\mu = r\mu]_m, \Theta$ contains $\mathrm{DefCond}(\mu, \Pi), \Pi[-]_n\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

`lemma-subs` $\textit{PS-Tree}\ \mu$

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle\ \ldots\ \langle \Lambda_n, \Gamma; w\rangle}$$

with $\langle \Pi; \hat{w}\rangle^{\mathcal{L}}$

if there is a substitution μ and a minimal clause Θ such that

- the goal labeling the root of the proof state tree *PS-Tree* is $\langle \Pi; \hat{w}\rangle$
- Γ, Θ contains $\mathrm{DefCond}(\mu, \Pi), \Pi\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

Figure 2.7: Non-Inductive Applicative Inference Rules Using Lemmas

literal as it cuts off the subgoal that otherwise would have to be created. A lemma is *directly applicable* to a goal if all condition literals are directly fulfilled in the goal. Definedness, condition and rewrite subgoals are created from left to right. This order is relevant insofar as we maximally downfold the definedness and the condition literals to the right.

`lemma-subs` is always applicable. Except for the rewrite subgoal, the application results in the same subgoals as for rewriting.

Inference rules `axiom-rewrite` and `axiom-subs` in Figure 2.8 do the same as inference rules `lemma-rewrite` and `lemma-subs` except that they apply axioms—i.e. defining rules—instead of lemmas. Whereas axioms are always inductively valid, lemmas that are applied in a proof attempt have to be checked for inductive validity to obtain a proof from a closed proof attempt.

The last non-inductive applicative inference rule removes redundant literals due to a lemma or axiom.

`axiom-rewrite` $m\ p\ \mathit{DefRule}\ n\ \mu$

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle\ \ldots\ \langle \Lambda_n, \Gamma; w\rangle\ \langle \Lambda, \Gamma[r\mu]_{m.p}; w\rangle}$$

with $\langle \Pi; ()\rangle^{\mathcal{L}}$

if there is a position $p \in \mathit{Pos}(\Gamma[m])$, a substitution μ and a minimal clause Θ such that

- the clause representation of the defining rule *DefRule* is Π
- $\Pi[n] = (l \doteq r)$
- $\Gamma[m]/p = l\mu$
- $\Gamma[l\mu = r\mu]_m, \Theta$ contains $\mathrm{DefCond}(\mu, \Pi), \Pi[-]_n\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

`axiom-subs` $\mathit{DefRule}\ \mu$

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle\ \ldots\ \langle \Lambda_n, \Gamma; w\rangle}$$

with $\langle \Pi; ()\rangle^{\mathcal{L}}$

if there is a substitution μ and a minimal clause Θ such that

- the clause representation of the defining rule *DefRule* is Π
- Γ, Θ contains $\mathrm{DefCond}(\mu, \Pi), \Pi\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

Figure 2.8: Non-Inductive Applicative Inference Rules Using Axioms

`appl-lit-removal` $m\ \mu\ \mathit{Lma}\ n$

$$\frac{\langle \Gamma; w\rangle}{\langle \Gamma[-]_m; w\rangle}$$

with $\langle \Pi; \hat{w}\rangle^{\mathcal{L}}$

if there is a substitution μ such that

- if *Lma* is a defining rule then Π is the clause representation of *Lma*
- if *Lma* is a proof state tree then the root of *Lma* is labeled with $\langle \Pi; \hat{w}\rangle$
- $\Gamma[m] =_{\mathrm{lit}} \overline{\Pi[n]\mu}$
- $\Gamma[-]_m$ contains $\mathrm{DefCond}(\mu, \Pi), \Pi[-]_n\mu$.

Figure 2.9: Non-Inductive Applicative Inference Rules for Removing Literals

`ind-rewrite` *m p PS-Tree n μ*

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle \ \ldots \ \langle \Lambda_n, \Gamma; w\rangle \ \langle \Lambda, \Gamma[r\mu]_{m.p}; w\rangle \ \langle \hat{w}\mu < w, \Lambda', \Gamma; w\rangle}$$

with $\langle \Pi; \hat{w}\rangle^{\mathcal{I}}$

if there is a position $p \in Pos(\Gamma[m])$, a substitution μ and a minimal clause Θ such that

- the goal labeling the root of the proof state tree *PS-Tree* is $\langle \Pi; \hat{w}\rangle$
- $\Pi[n] = (l \doteq r)$
- $\Gamma[m]/p = l\mu$
- $\Gamma[l\mu = r\mu]_m, \Theta$ contains $\mathrm{DefCond}(\mu, \Pi), \Pi[-]_n\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ
- $\Lambda' = \ l\mu = r\mu, \Lambda$.

`ind-subs` *PS-Tree μ*

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle \ \ldots \ \langle \Lambda_n, \Gamma; w\rangle \ \langle \hat{w}\mu < w, \Lambda, \Gamma; w\rangle}$$

with $\langle \Pi; \hat{w}\rangle^{\mathcal{I}}$

if there is a substitution μ and a minimal clause Θ such that

- the goal labeling the root of the proof state tree *PS-Tree* is $\langle \Pi; \hat{w}\rangle$
- Γ, Θ contains $\mathrm{DefCond}(\mu, \Pi), \Pi\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

Figure 2.10: Inductive Applicative Inference Rules

`appl-lit-removal` complements inference rules `lemma-subs` and `axiom-subs`. If the conjugate of the focus literal follows from the head literal and if all remaining condition and definedness literals are directly fulfilled, then the focus literal does not contain any additional information w.r.t. the remaining literals in the goal clause and the applied lemma (or axiom). Therefore, the focus literal can be removed from the clause safely.

2.2.2.11 Inductive Applicative Inference Rules

Lemmas can also be applied as induction hypothesis. For this, it has to be checked that the applied induction hypothesis is smaller w.r.t. the wellfounded induction order than the original goal it is applied to. Therefore, inference rules `ind-rewrite` and `ind-subs` in Figure 2.10 generate—in comparison to their non-inductive counterparts `lemma-rewrite` and `lemma-subs` in Figure 2.7—one additional *order subgoal* to the right. This order subgoal essentially extends the original goal by an order atom comparing the weights of the induction hypothesis and the original goal.

2.2.3 An Example Proof

We conclude the presentation of the logic of QuodLibet with an example to illustrate the introduced notions. In particular, we show how inductive proofs are performed using the inference system from Section 2.2.2, and how they are represented with proof state trees. The example is taken from our case study about the *lexicographic path order* LPO [KL80] (cf. Definition 8.1). In the case study we prove that the LPO is a *simplification order* on terms (cf. Definition 8.2). Additionally, we show the equivalence of different implementations of the LPO. The LPO example is challenging as the specification heavily depends on mutually recursive operators over mutually recursive data types. Therefore, the properties are proved by mutual induction which poses huge problems to many inductive theorem provers. QuodLibet is well suited for these specifications based on mutual recursion/induction because of its flexible inference system that allows for the *lazy* generation of induction hypotheses. By this we mean that we do not have to fix the induction hypotheses as well as the induction order at the beginning of the proof as in *explicit induction* but that we may apply lemmas inductively and have to fix the induction order only at the end of the proof. The flexibility QuodLibet provides to model the inductive proof process is explained more precisely in Section 3.1. In this section, we present a complete but small extract of the LPO example. An overview of the whole specification can be found in Chapter 8 and the complete proof script in Appendix A.

The LPO provides a scheme of wellfounded orders on wellformed terms over function symbols with fixed arity and variable symbols. It depends on a *precedence*, i.e. a quasi-order on function symbols. In our case study, we consider only total precedences. To model the LPO scheme within QuodLibet, we first have to specify a data type for (wellformed) terms. For this, we use a sort `Term`. The wellformedness property is modeled with a boolean valued function symbol. Therefore, we also require a sort `Bool` for boolean values. As we have to distinguish between terms starting with a function symbol and those starting with a variable symbol, the sort `Term` is defined with two constructors: constructor `V` generates a term from a variable symbol, constructor `F` constructs a term from a function symbol and a list of terms used as arguments. Thus, we get three additional sorts: sort `VID` represents the set of variable symbols, sort `FID` the set of function symbols, and sort `Termlist` represents lists of terms. We do not model many-sorted signatures in the LPO example, but we associate with each function symbol a natural number standing for its arity. Natural numbers are modeled with sort `Nat`. A term is wellformed iff the arity of its top-level function symbol corresponds to the length of its argument list and each term in its argument list is itself wellformed. In this section, we want to specify this property formally and prove its definedness on all terms, i.e. we can decide for each term whether it is wellformed. Altogether, our specification consists of the following sorts listed with their corresponding constructors:

`Bool` for boolean values with constructors

- `true` : $\rightarrow$ `Bool`
- `false` : $\rightarrow$ `Bool`

The two constructors are constants that represent the two truth values.

`Nat` for natural numbers with constructors

- $\texttt{0} : \to \texttt{Nat}$
- $\texttt{s} : \texttt{Nat} \to \texttt{Nat}$

The constructors represent zero and the successor function on natural numbers, respectively. Thus, $\texttt{s}(\texttt{s}(\texttt{0}))$ stands for the natural number 2.

`VID` for variable symbols with constructor

- $\texttt{Vid} : \texttt{Nat} \to \texttt{VID}$

The argument of the constructor is just used for distinguishing the variables. For each natural number, we get a different variable. Thus, the set of different variable symbols is countably infinite.

`FID` for function symbols with constructor

- $\texttt{Fid} : \texttt{Nat}, \texttt{Nat} \to \texttt{FID}$

The first argument of the constructor is used for calculating the precedence of the function symbols: a function symbol f is smaller in the precedence than a function symbol g iff this holds true for the first arguments of their constructors. The second argument represents the arity of the function symbol.

`Term` for terms with constructors

- $\texttt{V} : \texttt{VID} \to \texttt{Term}$
- $\texttt{F} : \texttt{FID}, \texttt{Termlist} \to \texttt{Term}$

Terms may be constructed from variable symbols or from function symbols together with a list of terms used as arguments.

`Termlist` for lists of terms with constructors

- $\texttt{nil} : \to \texttt{Termlist}$
- $\texttt{cons} : \texttt{Term}, \texttt{Termlist} \to \texttt{Termlist}$

Lists of terms are generated from the empty list `nil` and a constructor `cons` that adds one term to the front of a list of terms.

Note that terms and lists of terms mutually depend on each other. Therefore, the wellformedness property for terms is also specified with two function symbols: A function symbol `Well` on terms and a function symbol `Well_tl` on lists of terms. `Well_tl` is defined to be true for a list of terms iff each term in the list is wellformed. For the specification of the two function symbols, we require information about the arity of function symbols and the length of lists of terms represented with function symbols `arity` and `length`, respectively. Thus, we specify the following defined function symbols:

- $\texttt{arity} : \texttt{FID} \to \texttt{Nat}$
- $\texttt{length} : \texttt{Termlist} \to \texttt{Nat}$

- $\texttt{Well} : \texttt{Term} \to \texttt{Bool}$
- $\texttt{Well\_tl} : \texttt{Termlist} \to \texttt{Bool}$

For the formal specification of the defining rules we use the following constructor variables:

- $b : \texttt{Bool}$
- $m, n : \texttt{Nat}$
- $x, y, z : \texttt{VID}$
- $f, g, h : \texttt{FID}$
- $t, u, v : \texttt{Term}$
- $ts, us, vs : \texttt{Termlist}$

In the following, we present the defining rules for the defined function symbols in clausal form. The arity of a function symbol is given by the second argument of the constructor for function symbols (Axiom (2.1)).

$$\{ \texttt{arity}(\texttt{Fid}(n, m)) = m \} \tag{2.1}$$

The length of a list of terms is defined recursively: the length of the empty list is zero (Axiom (2.2)); the length of a non-empty list is the successor of the length of the remaining elements in the list without the first element (Axiom (2.3)).

$$\{ \texttt{length}(\texttt{nil}) = \texttt{0} \} \tag{2.2}$$

$$\{ \texttt{length}(\texttt{cons}(u, us)) = \texttt{s}(\texttt{length}(us)) \} \tag{2.3}$$

If the top-level symbol of the term is a variable, then the term is wellformed (Axiom (2.4)). If the top-level symbol is a function symbol and the arity of the function symbol corresponds to the length of the argument list, then the term is wellformed iff each of the terms in the argument list is wellformed (Axiom (2.5)). Otherwise, if the top-level symbol is a function symbol but the arity of the function symbol is unequal to the length of the argument list, then the term is not wellformed (Axiom (2.6)). The negative definedness literals are added to the defining rules to fulfill the admissibility conditions.

$$\{ \texttt{Well}(\texttt{V}(x)) = \texttt{true} \} \tag{2.4}$$

$$\begin{aligned} \{ & \texttt{Well}(\texttt{F}(f, ts)) = \texttt{Well\_tl}(ts), \\ & \texttt{arity}(f) \neq \texttt{length}(ts), \\ & \neg\texttt{def arity}(f), \\ & \neg\texttt{def length}(ts) \} \end{aligned} \tag{2.5}$$

$$\begin{aligned} \{ & \texttt{Well}(\texttt{F}(f, ts)) = \texttt{false}, \\ & \texttt{arity}(f) = \texttt{length}(ts), \\ & \neg\texttt{def arity}(f), \\ & \neg\texttt{def length}(ts) \} \end{aligned} \tag{2.6}$$

In an empty list of terms, each term is wellformed (Axiom (2.7)). If the first term of the list is wellformed then the whole list is wellformed iff this holds true for the list of the remaining terms without the first term (Axiom (2.8)). Otherwise, if the first term in the list is not wellformed, then the wellformedness property does not hold for all terms in the list (Axiom (2.9)).

$$\{\ \mathtt{Well\_tl}(\mathtt{nil}) = \mathtt{true}\ \} \tag{2.7}$$

$$\{\ \mathtt{Well\_tl}(\mathtt{cons}(u, us)) = \mathtt{Well\_tl}(us),\ \mathtt{Well}(u) \neq \mathtt{true}\ \} \tag{2.8}$$

$$\{\ \mathtt{Well\_tl}(\mathtt{cons}(u, us)) = \mathtt{false},\ \mathtt{Well}(u) = \mathtt{true},\ \neg\mathtt{def}\ \mathtt{Well}(u)\ \} \tag{2.9}$$

We want to prove that function symbol `Well` is defined for all terms (Lemma (2.10)). For its proof, we require a corresponding property for lists of terms, namely, that function symbol `Well_tl` is defined for all lists of terms (Lemma (2.11)).

$$\{\ \mathtt{def}\ \mathtt{Well}(t)\ \} \tag{2.10}$$

$$\{\ \mathtt{def}\ \mathtt{Well\_tl}(us)\ \} \tag{2.11}$$

For the proofs of Lemmas (2.10) and (2.11), we require the following lemmas: Function symbol `arity` is defined for all function symbols (Lemma (2.12)). Function symbol `length` is defined for all lists of terms (Lemma (2.13)). Their inductive proofs are trivial.

$$\{\ \mathtt{def}\ \mathtt{arity}(f)\ \} \tag{2.12}$$

$$\{\ \mathtt{def}\ \mathtt{length}(us)\ \} \tag{2.13}$$

The proofs for Lemmas (2.10) and (2.11) are represented with proof state trees in Figures 2.11 and 2.12, respectively. The root goal node consists of the conjecture to be proved and is displayed at the top of the proof state tree. Goal nodes are illustrated in rectangular boxes. They contain the clause of the goal in set notation with curly braces, and the weight of the goal in the last line of the goal node. Inference nodes are illustrated in rounded boxes. To keep the presentation concise, we do not present all the parameters of the inference rules. Only for the inference rules `subst-add` and `lit-add`, we add all the parameters, i.e. the substitutions and literals, respectively, that the case splits are based on. Furthermore, the inference rules for rewriting and subsumption refer to the lemmas[5] that they apply. The other parameters of the inference rules can be determined as follows: If the parameter of an inference rule refers to a literal in the goal clause this literal is underlined. For inference rules that apply lemmas for rewriting, we underline the subterm to be rewritten. This allows for the determination of the position required for the inference rule. With boxes we mark literals that are used as cut-off literals in the application of a lemma. From this information it is easy to complete the parameters of the inference rules such as the literal in the lemma clause that is used as head literal for rewriting or the matching substitution.

As we want to present closed proof state trees in the figures, the weight variables have already been instantiated. This need not be done at the beginning of the proof but before the order subgoals resulting from inductive applications are proved. In Figures 2.11 and

[5]In the following, we use the term "lemma" as generic term for both, axioms and lemmas.

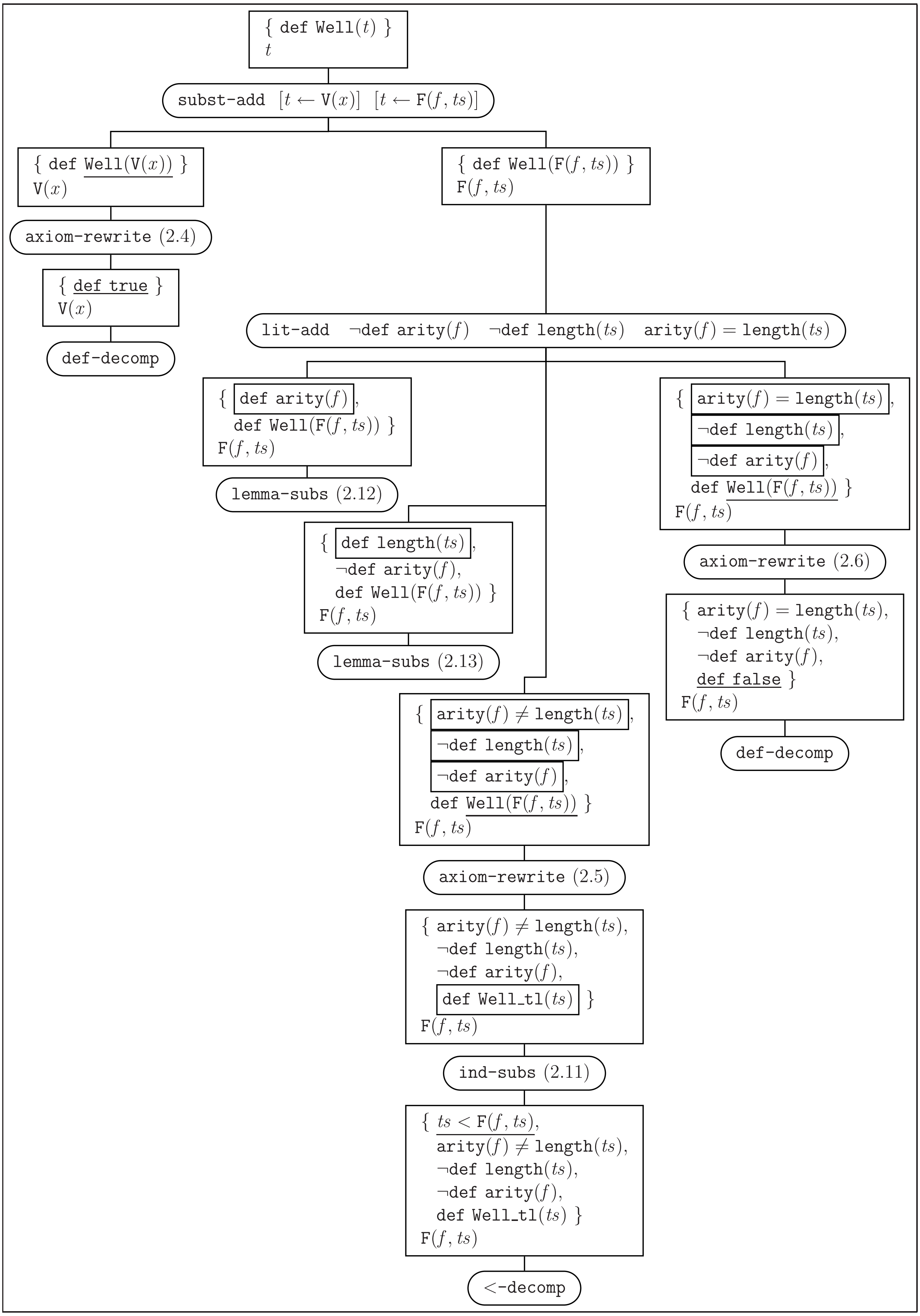

Figure 2.11: Proof State Tree for Lemma (2.10)

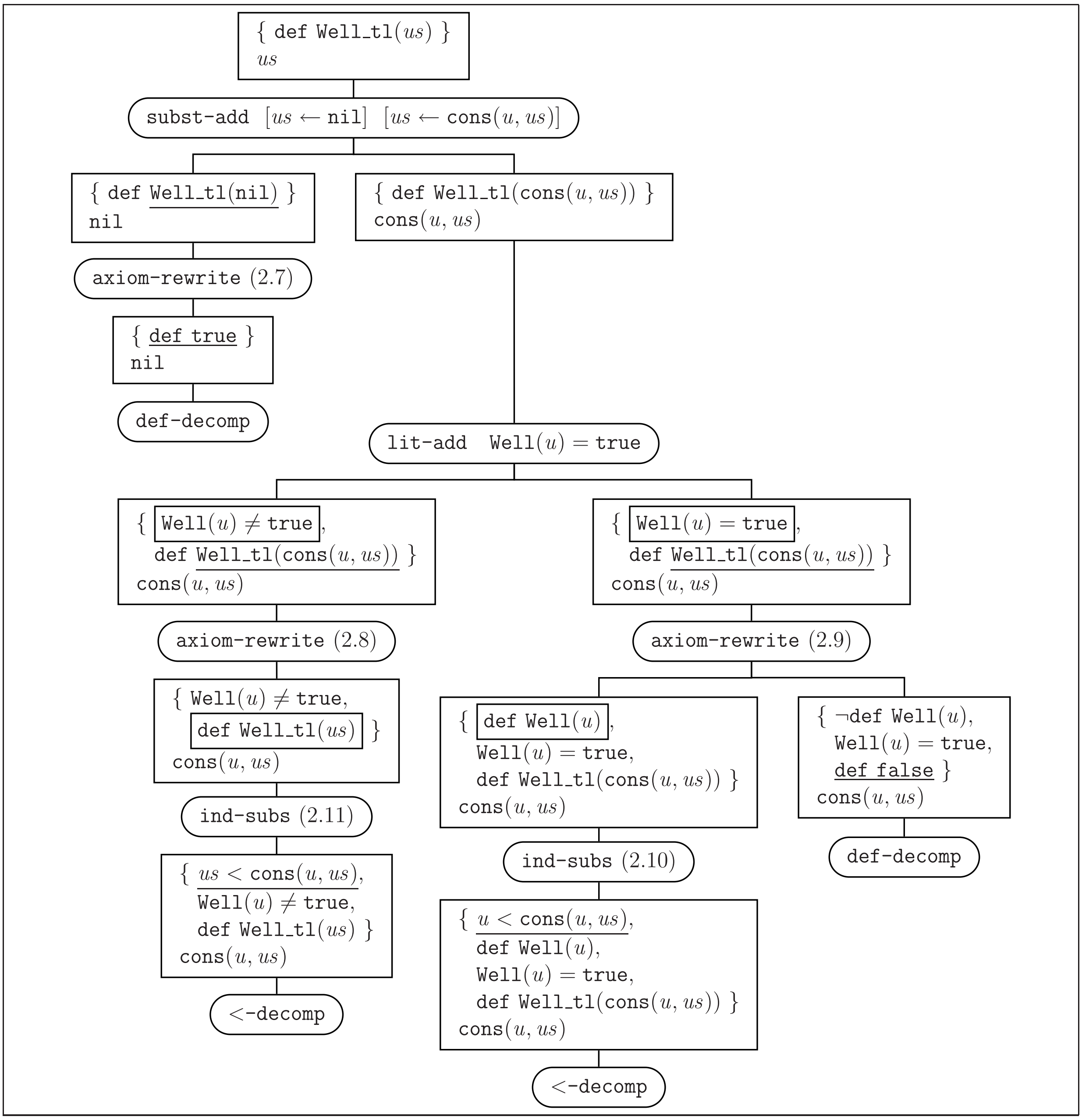

Figure 2.12: Proof State Tree for Lemma (2.11)

2.12, the lowest subgoals—that are generated by inference rule `ind-subs`—are the order subgoals. Before the instantiation, the weights of the two proof state trees are represented by two different weight variables: instead of weight t the proof for Lemma (2.10) starts with weight $w_{2.10}(t)$, and at the beginning of the proof for Lemma (2.11), weight us is replaced with weight $w_{2.11}(us)$ where $w_{2.10}$ and $w_{2.11}$ are free existential variables.

The figures present typical examples for inductive proofs performed with QUODLIBET: At first, an inductive case split is performed with inference rules `subst-add` and `lit-add` in such a way that the axioms of at least one defined operator can be applied. After that, the resulting subgoals are simplified as much as possible until no new subgoals are generated (in the base cases); or the lemma itself—or other mutually dependent lemmas—can be applied as induction hypotheses (in the step cases). If the proof process is successful, this results in some (unproved) order subgoals that are to guarantee the wellfoundedness of the induction scheme. After the induction order has been fixed by instantiating the weight variables associated with the proof state trees, these order constraints have to be proved to close the proof attempt.

In our example, the simplification process would have resulted in the following order constraints if the weight variables had not already been instantiated:

- $w_{2.11}(ts) < w_{2.10}(\mathtt{F}(f, ts))$
- $w_{2.11}(us) < w_{2.11}(\mathtt{cons}(u, us))$
- $w_{2.10}(u) < w_{2.11}(\mathtt{cons}(u, us))$

In this case, the weight variables can simply be instantiated by a projection on the first argument. This results in the order constraints shown in the proof state trees which can be closed immediately. As the proof state trees are closed, Lemmas (2.10) and (2.11) are inductively valid provided that this holds true for the applied Lemmas (2.12) and (2.13).

Certainly, the proof process may also stop unsuccessfully. This may be caused by an unsuitable case split or some missing auxiliary lemmas. In this case, the user has to provide additional information.

The proof state tree for Lemma (2.11) in Figure 2.12 is generated by the automatic proof control presented in this thesis. Lemma (2.10) is also proved automatically but the proof state tree that is generated automatically is slightly more complicated than that presented in Figure 2.11: In the case split with inference rule `lit-add` only equality literals are considered in the automatically generated proof attempt. Subgoals similar to the following two left-most subgoals starting with the definedness atoms $\mathtt{def\ arity}(f)$ and $\mathtt{def\ length}(ts)$, respectively, are created twice as condition subgoals for the rewrite steps with Axioms (2.5) and (2.6) (cf. the application of Axiom (2.9) in Figure 2.12). Thus, the automatic proof control creates two additional subgoals.

2.3 The Architecture

In this section, we give a brief overview of the system architecture of QUODLIBET. Further information can be found e.g. in [Küh00] and [Kai02]. In particular, we do not present any details about the command language or the graphical user interface.

The system architecture is illustrated in Figure 2.13. QuodLibet is designed in a modular way on three levels:

- The basic level realizes an *inference machine kernel*. It is responsible for the compliance with the logic presented in Section 2.2. In particular, it ensures the admissibility of the given specification. Since the proof state graph can be altered only by applying inference rules implemented in the kernel, the inductive validity of the proved lemmas is guaranteed—regardless whether the proof is done manually or automatically—provided that the implementation of the kernel is error-free.

- On the highest level, there are two user interfaces for the communication with the system: On the one hand, a text-based user interface is realized by a simple command language. This language is also used for saving proof scripts to files. In this way, we can establish libraries for specifications and proofs. On the other hand, there is a graphical user interface XQL that is an add-on for the text-based interface, i.e. the actions of XQL are translated into commands of the text-based interface. In this way, XQL realizes most commands of the command language with graphical shortcuts that are easy to use. Furthermore, XQL provides a sophisticated back-end for presenting proof state trees which makes the analysis of failed proof attempts a lot easier.

- An optional intermediate level allows for the automation of the proof process. It essentially consists of a compiler that translates routines written in an adapted imperative programming language called QML (QuodLibet meta language) into executable code. The code of the public routines may be called via the text-based user interface just like the inference rules of the kernel. For ease of use, the routines are automatically integrated into the graphical user interface as well. As it is the most important level for this thesis, an overview of QML is given in Section 2.3.1.

The command language allows the user to

- initialize the inference machine kernel;
- enter specifications and lemmas;
- apply inference rules and instantiate weight variables manually;
- navigate in proof state trees;
- delete (subtrees of) proof state trees;
- load and save proof scripts;
- compile QML code for the automation of proof control;
- call public routines from the automatic proof control; and
- display all kinds of information such as specifications, proof state trees and their dependencies which are managed automatically.

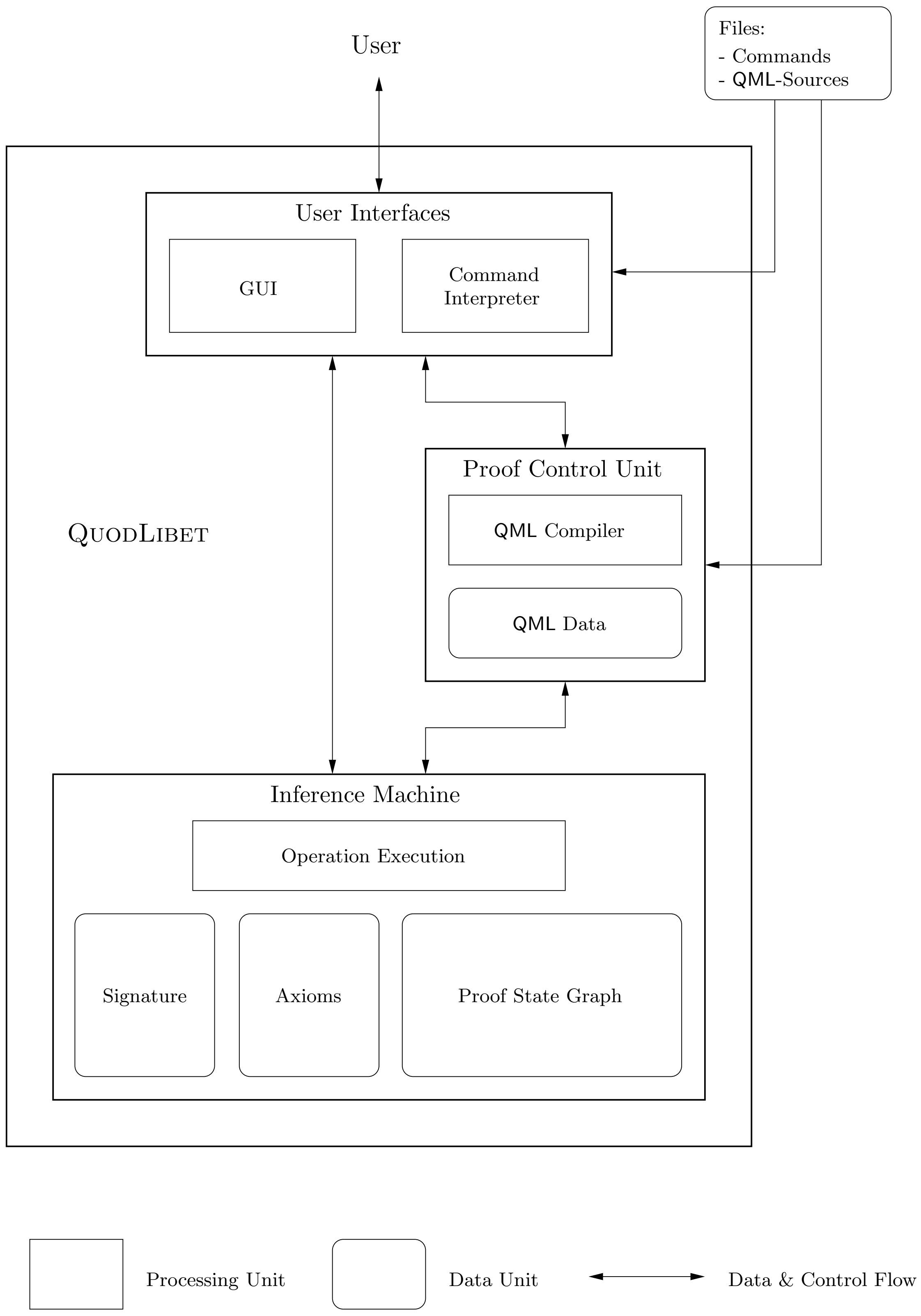

Figure 2.13: High Level System Structure of QuodLibet

2.3.1 Tactic-Based Proof Control

In comparison to a "hard-wired" proof control, a tactic-based proof control provides much more flexibility. Tactics allow the user to combine applications of inference rules in a flexible way using a programming language. Tactics may be interpreted as derived inference rules that handle a certain proof pattern. Often, the functional programming language ML [Pau96] is used for implementing tactics. In contrast to this, QuodLibet provides its own adapted imperative programming language QML for this task due to the following reasons: For implementing control, an imperative programming seems to be more adequate than a functional one. It provides the usual control structures such as loops and conditional statements. On the one hand, the similarity of QML to PASCAL eases the usage of QML for many programmers. On the other hand, the adaptation to QuodLibet provides them with an easy access to the relevant data structures of the prover. Furthermore, the adaptation allows to restrict the programming language to the required concepts. This facilitates both, to port tactic code to other (subsequent) provers, as well as, to add new concepts or change previous concepts during system evolution.

In this section, we summarize the basic concepts of QML. The QML compiler was designed and implemented based on concepts developed in [Spr96]. A detailed description of QML and the realization of the compiler can be found in [SS97]. In the meanwhile, QML has been modified only to a little extent. Most notably, another type constructor for hashes has been introduced. Furthermore, the library of predefined routines provided by the kernel has been extended. For instance, it is now possible not only to apply inference rules automatically but also to delete subtrees of proof state trees during an automatic proof attempt. As a result, some concepts for tactics could not be realized efficiently anymore and have been abandoned: Tactics do not fail automatically anymore even if they do not alter the proof state graph. The mechanism of applying automatically a list of tactics—associated with a tactic T—to each open subgoal that is generated during a call to tactic T has been eliminated.

2.3.1.1 Modules

QML allows for the structuring of code into modules using the principle of *information hiding*. Each *module* is implemented in a single file and provides its own *namespace*, i.e. the same name in two different modules may be used for two different entities. This enables the development of large QML-programs for proof control implemented by different users.

Entities that are to be shared between different modules have to be declared in the import/export interfaces of the modules. Tactics and procedures may be called from the user interface of QuodLibet only if they are made publicly available in the export interface.

2.3.1.2 Data Types

QML is a *typed* language with *static* type checking as far as possible. Thus, for each variable and for each routine, its type has to be declared. QML provides some *basic types* as well as *type constructors* to build new composite types.

Beside standard types for integers, reals, strings and boolean values, QML possesses predefined types for all important data structures of the inference machine kernel such as sorts and operators to access signatures; terms, literals, formulas, substitutions, positions and weights for the so-called *prover objects*; goal and inference nodes to handle proof state trees. Primitive routines for these data types are made available through a library which can be easily extended.

Furthermore, QML provides type constructors for building *lists* of elements of the same type, *structures* of elements with possibly different types, *enumeration* types, and *hashes* mapping one primitive type such as integers, strings or goal nodes to another type.

2.3.1.3 Routines

The executable part of the code is structured in *routines*. QML distinguishes three types of routines: procedures, functions, and tactics. Procedures and functions are well-known from other programming languages. Whereas *procedures* never return a value, *functions* always return a value. *Tactics* are special forms of procedures. Only within a tactic, proof state trees may be changed by applying inference rules or deleting subtrees. As tactics are considered as derived inference rules, for each tactic, the first parameter is mandatorily a goal node the tactic is applied to.

2.3.1.4 Statements

The body of a routine is composed of *statements*. QML provides statements for

- calling procedures and tactics;
- assigning values to variables;
- repeating statements with `for`, `foreach`, `while`, `repeat-until` and general `loop` constructs with explicit `break` statements;
- executing statements conditionally with `if-then-else` constructs;
- returning from a loop or routine immediately; and
- handling exceptions with a `try` and `fail` mechanism.

2.3.1.5 Expressions

Expressions are used for computing values. QML provides the usual constructs for

- listing constants of standard types;
- accessing (sub-)values of (composed) variables;
- calling functions; and
- applying predefined operators such as arithmetic operators, comparison operators, boolean connectives and operators working on lists.

Furthermore, there are special expressions e.g. for constructing prover objects such as terms, literals, formulas and substitutions; and for matching prover objects to patterns containing QML-variables that are automatically bound if a match exists.

In the following chapters, we illustrate some of our concepts with code fragments. Not to overwhelm the reader with too many technical details, we do not present these code fragments in QML but in a simplified pseudo code that omits e.g. the declaration of variables. This pseudo code, however, can be easily translated into QML.

Chapter 3

Automating Inductive Theorem Proving

In the literature, there exist many different proof search techniques for automating inductive proofs. [Bun01] contains a survey based on seminal work done in [BM79], [BM88a] and [BM88b]. Some of the proof search techniques are not restricted to inductive theorem proving only, such as simplification techniques that

(1a) apply lemmas for rewriting or subsumption,

(1b) use equality information for cross fertilization, and

(1c) employ decision procedures.

Others are special to inductive theorem proving, such as finding suitable

(2a) induction schemes,

(2b) generalizations of lemmas, and

(2c) intermediate lemmas.

Whereas some proof search techniques, such as (1a)–(1c) and (2a), can often be automated in a suitable way based on syntactical information, the speculation of auxiliary lemmas in (2b) and (2c) usually requires the support of an experienced user with domain knowledge. In this thesis, we attempt to improve the automation of inductive theorem provers in combination with a suitable form of user-interaction.

- In Chapter 4, for instance, we investigate whether the close integration of decision procedures can enhance the speculation of auxiliary lemmas.
- Most work in the automated proof process is caused by the application of (conditional) lemmas. In our case studies, they cause at least 50% of all successful proof steps. In Chapter 6, we present novel heuristics that are particularly useful for this important task. The efficiency and extent of our novel heuristics may be influenced manually in an easy way.

In this chapter, we present the top-level proof control used for the automation. For the integration of the different search techniques into a single proof process, successful inductive theorem provers such as `NQTHM` and `ACL2` use a waterfall model. In Section 3.2, we refine a simple waterfall (3.2.1) to a more flexible one (3.2.2). In Chapter 7, we enhance the simple waterfall even more by complementing the proof search techniques with reuse techniques.

One of the most distinguishing features among inductive theorem provers is concerned with the integration of induction schemes into the proof process. In Section 3.1, we present a brief survey of the different integration techniques such as proof by consistency (3.1.1), explicit induction (3.1.2) and descente infinie (3.1.3). For the automation of QUODLIBET, we use an inductive proof process based on descente infinie because this process is most suitable for supporting user-interaction. The skeleton for integrating descente infinie into QUODLIBET is retained from the former proof control described in [Küh00] based on ideas from [BM79] and [Wal94]. An improved version is sketched in Section 3.1.4, more details can be found in [SS04].

3.1 Inductive Theorem Proving

The induction scheme determines the base and step cases into which the proof is split. Furthermore, it identifies the induction hypotheses that can be used for proving the induction conclusion in the step case. It has to be proved that the induction hypotheses are smaller than the induction conclusion w.r.t. a wellfounded induction order. The following methods differ in the way how and when they acquire the needed information.

Suitable induction schemes can often be computed automatically using recursion analysis techniques. Nevertheless, for complicated proofs user-interaction may be required as well. Therefore, we prefer those methods that allow the user to provide the needed information manually in an easy way.

3.1.1 Proof by Consistency

Proof by consistency is a method based on completion techniques. In the purely equational case, an unfailing Knuth-Bendix completion procedure may be used as described in [Mus80] and [Bac88]. An extension for arbitrary clauses with equality based on a superposition calculus can be found in [GS92].

In [Bac88], the axioms are given as a *ground convergent* rewrite system R, i.e. the induced rewrite relation is confluent and terminating on ground terms. An equation is *consistent* with a specification if every ground instance is joinable, i.e. both sides of the instance are reducible to the same term. In this case, the equation is inductively valid. To prove consistency, the absence of any inconsistencies is shown. The approach is based on an inference system that reduces inconsistencies to smaller ones—e.g. by adding critical pairs of rewrite rules in R on equations to be proved—until a *provably inconsistent* equality is derived. The absence of inconsistencies is guaranteed for *fair* derivations of the inference system that terminate without generating any provably inconsistent equalities (cf. [Bac88] for details).

The method is also named *inductionless induction* in [Com01] because it does not make use of explicit induction rules. But this also complicates the manual provision of induction schemes. In [Wir05b], the disadvantages of the approach are summarized as follows:

> '[It] produces a high number of irrelevant inferences under which the ones relevant for induction can hardly be made explicit in an automatic way. "Inductionless induction" has shown to be practically useless, mainly due to too many superfluous inferences, typically infinite runs, and too restrictive admissibility conditions. [...] Roughly speaking, the conceptual flaw in "inductionless induction" seems to be that, instead of finding a sufficient set of reasonable inferences, the research follows the paradigm of ruling out as many irrelevant inferences as possible.'

Therefore, we do not consider this approach furthermore.

3.1.2 Explicit Induction

Explicit induction is used in many inductive theorem provers such as `NQTHM` [BM88a], `ACL2` [KMM00], and `RRL` [KZ95]. The method is described, for instance, in [Wal94] and [Bun01].

In explicit induction, a deductive inference system is enhanced with one inductive rule that introduces an induction scheme. An application of the inductive rule performs a case split, generates the induction hypotheses that can be used in the step cases, and proves the wellfoundedness of the induction scheme *at once*. For each induction step, an implication is constructed relating induction hypotheses to induction conclusions. No further induction hypotheses may be used in the induction step. The induction schemes can be generated only according to (combinations of) terminating total function definitions that guarantee the wellfoundedness of the associated induction order. The automatic application of the inductive rule depends on a strong recursion analysis process. Recursion analysis is, for instance, described in [BM88a] and, under the name *cover set method*, in [ZKK88].

Explicit induction can take advantage of the early introduction of the induction hypotheses if they are appropriate:

- The simplification process can be guided in a goal-directed way using heuristics such as rippling techniques (cf. [BSvH$^+$93, Hut97]).
- The application of lemmas, as e.g. transitivity lemmas, can be supported by providing additional information about the instantiation of free variables (cf. Example 3.3 in [Wir04]).
- The information can be used for the generalization of lemmas and the speculation of additional lemmas (cf. [BM88a]).

But there are some problems with the explicit induction approach as well: As explained in [Pro94] it is not always possible to compute appropriate induction hypotheses at the beginning of a proof attempt. Thus, the method of explicit induction may fail even for relatively simple examples unless the user provides an appropriate induction scheme himself. In `NQTHM` and `ACL2` this can be done by indicating the defined operator whose recursive definition scheme is to be used as induction scheme. This often leads to the definition of dummy operators just to provide an appropriate induction scheme. Especially, when dealing with mutually recursive functions this seems to raise great difficulties.

3.1.3 Descente Infinie

In *descente infinie* (cf. [Wir04]), it is not necessary to provide the whole induction scheme in a single step as in explicit induction. Instead, the information may be given in three different steps. Therefore, we may fix the induction hypotheses and the induction order as late as needed. This approach is similar to *lazy induction* described in [Pro94].

To be more precise, an inductive proof for a clause Γ performed by descente infinie can be divided into the following steps:

1. At first, a case analysis is performed leading to a case split. This step may be performed automatically according to the recursion analysis of the operators in Γ using similar techniques as for explicit induction [BM88a, ZKK88] without fixing the induction hypotheses.

2. Each case is simplified by applying inference rules and using the given lemmas in order to reduce it to a valid formula (base case) or to apply a smaller instance of Γ (induction step) or other lemmas that are used by mutual induction.

3. It has to be shown that only smaller instances are used in the induction step. Therefore an appropriate wellfounded induction order has to be selected and the order constraints have to be proved.

On the one hand, the eager generation of induction hypotheses may be advantageous for the automation of the simplification process and the speculation of auxiliary lemmas as described in Section 3.1.2. Therefore, the lazy generation of induction hypotheses in descente infinie may result in a slightly less automated proof process. In [Pro94], however, rippling techniques are modified in such a way that they are independent from eager hypotheses generation: Instead of using rippling techniques to rewrite an induction conclusion to a concrete induction hypothesis it is only important to move the differences of the induction conclusion (w.r.t. the original lemma) to tolerable positions, i.e. to top-level or variable positions.

On the other hand, a proof process based on descente infinie provides much more flexibility. Each single step may be performed independently. Therefore, it is much easier to acquire the needed information regardless of whether this is done automatically or manually. The benefits can be illustrated best for mutually recursive operators: Proofs of properties for mutually recursive operators often have to be performed by mutual induction. This causes difficulties in speculating auxiliary lemmas for each recursive operator and generating appropriate induction schemes for each lemma. The complexity of the latter can be reduced by choosing the inductive case split, the induction hypotheses and the induction order independently as soon as the needed information is available. Often, the inductive case split is performed in such a way that the axioms of some defined operators are applicable. But this information is independent of the recursive dependencies. Therefore, it may be acquired for mutually recursive operators automatically as easily as for non-mutually recursive operators. The same holds true for the simplification process except that, of course, the step cases that have to use properties of mutually dependent operators cannot be closed. But often the resulting subgoals support the user in speculating the needed auxiliary lemmas. Therefore, we get a synergetic effect resulting from the interplay between automation and user-interaction.

This approach for proving properties of mutually recursive operators may be illustrated with the simple example proofs of Lemmas (2.10) and (2.11) in Section 2.2.3 that are depicted in Figures 2.11 and 2.12, respectively. If we try to prove the lemmas independently of each other, the proof attempts will stop exactly for those two subgoals that the other lemma is applied to by inference rule `ind-subs`. But these subgoals provide enough information for speculating the corresponding lemmas easily. This approach has been extensively used for performing the case study about the LPO (cf. Chapter 8).

Even if the case split cannot be generated automatically, it may be given by the user more easily. The user does not have to define dummy operators to provide an induction scheme as in explicit induction. Instead, the case split may be introduced directly. The user does not even have to worry about induction hypotheses in this step at all. He can simply start the simplification process for the subgoals resulting from the case split, see whether the required induction hypotheses are applied automatically, and analyze failed proof attempts.

With descente infinie, we may use induction schemes that do not perform an inductive case split based on the recursive structure of defined operators explicitly. Instead, we may derive smaller instances by applying lemmas. This is illustrated in the following example taken from our case study that $\sqrt{2}$ is irrational with a proof based on ideas from Euclid of Alexandria. The lemma is proved automatically with our proof control based on descente infinie.

Example 3.1 For this example, we assume sorts `Bool` and `Nat` to be defined as in Section 2.2.3. Furthermore, we require the following defined operators

- $\texttt{not}: \texttt{Bool} \rightarrow \texttt{Bool}$
- $\texttt{even}: \texttt{Nat} \rightarrow \texttt{Bool}$
- $\texttt{half}: \texttt{Nat} \rightarrow \texttt{Nat}$
- $\texttt{+}: \texttt{Nat}, \texttt{Nat} \rightarrow \texttt{Nat}$
- $\texttt{*}: \texttt{Nat}, \texttt{Nat} \rightarrow \texttt{Nat}$

defined with axioms

$$\{\ \texttt{not}(\texttt{true}) = \texttt{false}\ \} \tag{3.1}$$
$$\{\ \texttt{not}(\texttt{false}) = \texttt{true}\ \} \tag{3.2}$$

$$\{\ \texttt{even}(\texttt{0}) = \texttt{true}\ \} \tag{3.3}$$
$$\{\ \texttt{even}(\texttt{s}(m)) = \texttt{not}(\texttt{even}(m))\ \} \tag{3.4}$$

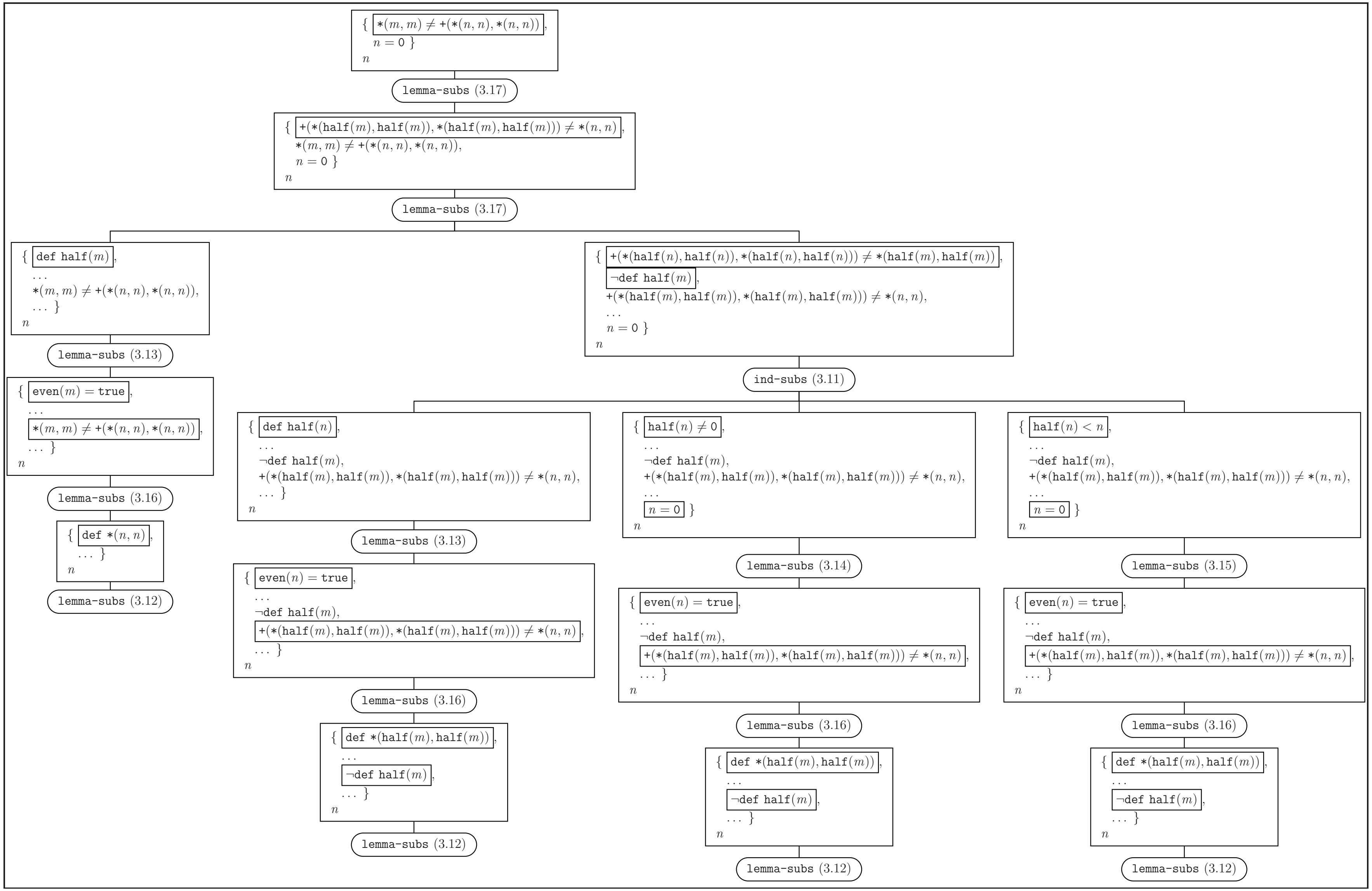

Figure 3.1: Proof State Tree for Lemma (3.11)

$$\{\ \mathtt{half}(0) = 0\ \} \tag{3.5}$$
$$\{\ \mathtt{half}(\mathtt{s}(\mathtt{s}(m))) = \mathtt{s}(\mathtt{half}(m))\ \} \tag{3.6}$$

$$\{\ \mathtt{+}(m, 0) = m\ \} \tag{3.7}$$
$$\{\ \mathtt{+}(m, \mathtt{s}(n)) = \mathtt{s}(\mathtt{+}(m, n))\ \} \tag{3.8}$$

$$\{\ \mathtt{*}(m, 0) = 0\ \} \tag{3.9}$$
$$\{\ \mathtt{*}(m, \mathtt{s}(n)) = \mathtt{+}(\mathtt{*}(m, n), m)\ \} \tag{3.10}$$

Note that operator `half` is defined only for even natural numbers resulting in a partial definition.

To prove the irrationality of $\sqrt{2}$ it is sufficient that, for each pair m, n of natural numbers (with $n \neq 0$), $m^2/n^2 \neq 2$, or equivalently $m^2 \neq 2n^2$. This may be expressed with the following lemma:

$$\begin{aligned}\{\ &\mathtt{*}(m, m) \neq \mathtt{+}(\mathtt{*}(n, n), \mathtt{*}(n, n)),\\ &n = 0\ \}\end{aligned} \tag{3.11}$$

Informally, this may be proved by contradiction: Assume that $m^2 = 2n^2$. Then, m is even (cf. Lemma (3.16)). Thus, $m/2$ is defined and $2(m/2)^2 = n^2$ (cf. Lemma (3.17)). With the same argument, we derive $(m/2)^2 = 2(n/2)^2$. But this is a smaller instance of the assumption w.r.t. to a wellfounded order (cf. Lemma (3.15)) leading to a contradiction. For a formal proof, we additionally require definedness properties for operators `*` (cf. Lemma (3.12)) and `half` (cf. Lemma (3.13)) as well as the property that $m/2$ is positive if it is defined and $m \neq 0$ (cf. Lemma (3.14)).

$$\{\ \mathtt{def}\ \mathtt{*}(m, n)\ \} \tag{3.12}$$
$$\begin{aligned}\{\ &\mathtt{def}\ \mathtt{half}(m),\\ &\mathtt{even}(m) \neq \mathtt{true}\ \}\end{aligned} \tag{3.13}$$
$$\begin{aligned}\{\ &\mathtt{half}(m) \neq 0,\\ &\mathtt{even}(m) \neq \mathtt{true},\\ &m = 0\ \}\end{aligned} \tag{3.14}$$
$$\begin{aligned}\{\ &\mathtt{half}(m) < m,\\ &\mathtt{even}(m) \neq \mathtt{true},\\ &m = 0\ \}\end{aligned} \tag{3.15}$$
$$\begin{aligned}\{\ &\mathtt{even}(m) = \mathtt{true},\\ &\mathtt{*}(m, m) \neq \mathtt{+}(n, n)\ \}\end{aligned} \tag{3.16}$$
$$\begin{aligned}\{\ &\mathtt{+}(\mathtt{*}(\mathtt{half}(m), \mathtt{half}(m)), \mathtt{*}(\mathtt{half}(m), \mathtt{half}(m))) = \mathtt{*}(n, n),\\ &\mathtt{*}(m, m) \neq \mathtt{+}(\mathtt{*}(n, n), \mathtt{*}(n, n))\ \}\end{aligned} \tag{3.17}$$

The resulting proof state tree for Lemma (3.11) is sketched in Figure 3.1. Due to lack of space, we omit literals in the goal clauses that are not used in the subsequent proof attempt anymore. Omitted literals are indicated with ellipsis "…". The proof state tree resembles the informal argumentation by first applying Lemma (3.17) twice, followed by one application of the induction hypothesis. Note that neither an inductive case split is introduced explicitly nor a single axiom is applied in the proof.

The proof of the main Lemma (3.11) is quite simple provided that the auxiliary Lemmas (3.12) to (3.17) are given. It is generated automatically by our proof control. The main difficulty arises from having to speculate appropriate auxiliary lemmas. Lemma (3.17), for instance, depends on operator `half` which is not present in Lemma (3.11). Therefore, the lemma can hardly be speculated automatically but it has to be supplied by a knowledgeable user who is familiar with the application domain. However, it is not difficult to derive the auxiliary lemmas from the informal proof. For further examples on the speculation of auxiliary lemmas, we refer to Section 8.2.2.

A minor problem consists in proving the auxiliary lemmas themselves as they make use of arithmetic properties. For their proofs, we require additional auxiliary lemmas. Furthermore, we apply two inference rules manually. In Chapter 4, we try to lessen this problem by integrating a decision procedure for linear arithmetic into QUODLIBET.

In [Wie03], the proof of the irrationality of $\sqrt{2}$ is used as a challenging problem for comparing 15 different theorem provers w.r.t. their ability to formalize and prove mathematics. The systems are evaluated w.r.t. "the size of their library, the strength of their logic and their level of automation". For the proof development system ΩMEGA [SBB+02], a more detailed description of this case study can be found in [SBF+03], allowing for a comparison with our approach. There, three different approaches are considered for the automation of the proof: an interactive proof where ΩMEGA is used as usual tactical theorem prover; a proof based on interactive "island planning"; and a fully automatically planned proof.

In ΩMEGA, different levels of abstractions may be considered: Proofs may be planned at a high level of abstraction; plans have to be expanded into lower levels of abstraction until, finally, a proof at the level of the logical calculus is established. The difference between the first two approaches is the initial level of abstraction resulting in different orders in which the proof steps on the lower levels are performed: In the tactical approach, the proofs are performed directly on the lowest level one after the other; in the island planing approach, first, the whole proof plan is generated on an abstract level, then, the gaps are filled on the lower levels. This results in a top-down approach for performing the proof. Note that both approaches may be modeled in QUODLIBET. The information supplied in the island steps may be used for speculating auxiliary lemmas. If we decide to prove the auxiliary lemmas immediately, we get the tactical approach. If we delay the proofs of the auxiliary lemmas until the main lemma is proved, we get the island planning approach.

In the third approach, domain knowledge is encoded in a predefined lemma base and the tactics of the proof control themselves. Thus, it is a specialized approach restricted to a special application domain. In contrast to this, we are interested in general approaches for proof control. Domain knowledge is merely encoded in auxiliary lemmas which is sufficient for this example. □

3.1.4 The Realization in QUODLIBET

With the inference rules of QUODLIBET (cf. Section 2.2.2), it is possible to realize both—explicit induction and descente infinie. For descente infinie, this may be done in the following way:

- Step 1 may be realized with inference rules `subst-add` and `lit-add` to introduce case splits according to constructor and destructor recursion, respectively.

- Whereas the simplification process in Step 2 may use all inference rules, induction hypotheses may be applied with inference rules `ind-subs` and `ind-rewrite` resulting in additional order subgoals to guarantee the wellfoundedness of the induction order.

- Because of the weight variables the induction order need not be fixed at the beginning of an inductive proof attempt in accordance with Step 3. It may be instantiated manually or automatically using a special command `set weight` of the command language of QuodLibet. After the instantiation of the weight variables, all inference rules may be applied to prove the order subgoals. Certainly, those inference rules concerned with order atoms are particularly suitable.

For the modeling of explicit induction, inductive case splits with inference rules `subst-add` and `lit-add`, the application of induction hypotheses with inference rules `ind-subs` and `ind-rewrite`, the instantiation of the weight variables with command `set weight` to fix the induction order, and the verification of the order constraints is performed at the beginning of an inductive proof attempt. But as we are interested in complicated inductive proofs that require an interplay of automation and user-interaction, and since this interplay is best supported by descente infinie, we concentrate on this proof process in the rest of this thesis.

A sensible approach for structuring the inductive proof process based on descente infinie into QML-modules is described in [Küh00]. Beside some auxiliary modules, it consists of the following modules that provide public routines:

`Database`: The database stores all information about *analyzed* operators and *activated* lemmas. The information about operators is used for creating the initial case split in an inductive proof. Only the activated lemmas are used for rewriting and subsumption during the simplification process. With the analysis of operators and the activation of lemmas, the user can guide the proof process on *various* levels. During the activation of a lemma, for instance, the user may fix those literals that may be used as head literals (cf. Section 2.2.2.10).

`Inductive-Case-Analyses`: This module contains routines for performing the inductive case split from Step 1. Additionally, it applies the axioms of the defined operators that the case split is based on.

`Simplification`: The routines in this module are responsible for simplifying goals as required for Step 2—including the application of lemmas as induction hypotheses—and proving the order constraints in Step 3.

`Proof-Strategies`: This module implements the whole proof process using the public routines from the other modules.

We have refined the structuring of the modules and improved their realization in [SS04]. In this thesis, we concentrate on some of the improvements of the simplification process that promise the highest profit. We give only a short summary of further improvements here. For a detailed description we refer to [SS04].

- Regarding Step 1, the analysis of defined operators for creating an inductive case split for a clause Γ *automatically* is already described in [Küh00]: The *expandable* operator calls of Γ, that do not *obstruct* other operator calls in the clause, are merged into one inductive case split. Our improvements enable the user to perform the case split *semi-automatically* by selecting the operator calls that should be considered or *manually* by specifying the induction variables themselves. In the latter case, an induction variable of sort s—defined with n constructors $c_1, \ldots, c_n$—is instantiated with n different terms in a canonical way: For each constructor c_i, one term is generated with top-level symbol c_i and pairwise different new[1] constructor variables as arguments. For the specification presented in Section 2.2.3, variable ts of sort `Termlist`, for instance, is instantiated with `nil` and $\texttt{cons}(t, ts)$ where t is a new variable. If more than one induction variable is selected all possible combinations are considered.

- The simplification process in Step 2 is completely reorganized in comparison to the approach in [Küh00] providing much more flexibility by a *parameterized* proof process. An even more improved organization is presented in Section 3.2. Furthermore, the application of conditional and permutative lemmas, the application of lemmas as induction hypotheses, the handling of order and negated equality atoms, the provision of alternative literal representations and the avoidance of repetitions of equal inference steps are improved.

- The instantiation of weight variables in Step 3 is improved by the introduction of *weight modifiers*. These enable the automatic proof control, for instance, to set the weight for sorting algorithms to the *length* of the sorted list instead of the list itself which leads to simpler proofs of the order constraints.

- The tactics that model the whole proof process are called *strategies*. A strategy is *recursive* iff it starts another inductive proof when it gets stuck. The two strategies in [Küh00] perform an automatic inductive case split, simplify the resulting goals possibly using the lemma to be proved as induction hypothesis. The strategies differ only in whether they are recursive. Our improved strategies, additionally, enable the user to choose whether the case split is to be performed automatically, semi-automatically or manually, leading to six different strategies. Furthermore, the user may activate lemmas in such a way that they may be used as induction hypotheses.

The refined structuring introduces in particular the following two new modules:

`Default-Settings:` The user may influence the proof process to a large extent with (optional) parameters. If the user does not supply a value for an optional parameter when calling a tactic a default value will be used. To provide more flexibility, these global default values themselves may also be specified by the user. This mechanism is realized in this module.

`Protected-Inference-Machine:` This module provides wrapper functions for the inference rules which add further functionality to each inference rule in a uniform way:

[1]New means that the variables must not occur in the instantiated clauses. Yet, the variable that is instantiated may be used again.

- The repeated application of the same inference rule in one path of a proof state tree or the same top-level application as in an alternative proof path is prevented.
- Literals in goal clauses may be marked to restrict proof search (cf. Chapter 6).
- We provide a *debug mode* that behaves exactly in the same way as the regular mode but does not physically delete failed proof attempts. Therefore, debug mode supports the user in analyzing failed proof attempts.

3.2 Top-Level Proof Control

In this section, we present *waterfall models* that are used for integrating different proof search techniques into a single proof process: The goals to be proved are collected in a *pool*. The proof techniques are grouped in phases. A *phase* may be interpreted as a derived inference rule: If a phase is applied to a goal successfully, the goal is reduced to new subgoals (which will not be deleted during the automatic proof attempt anymore). Otherwise, the phase must not change the proof state tree. More precisely, the phase may apply inference rules in the meantime but, finally, it has to restore the previous state of the tree. Thus, it has to delete all proof steps applied by itself. Each goal passes through each phase successively until one of them can be applied successfully. The subgoals are put into the pool again starting the proof process from the beginning. Thus, we get a recursive control structure for the proof process. The idea of a waterfall model is to start with the cheapest phases that promise the highest profit.

In `NQTHM` and its successor `ACL2`, the whole inductive proof process is structured in a single waterfall. In contrast to this, we implement the inductive proof process as described in Section 3.1.4 and use a waterfall model to structure only the simplification process. This can be motivated as follows: In provers based on explicit induction, induction schemes are solely introduced with inductive rules and do not pose any restrictions on the applications of other inference rules. Therefore, the phases are independent from each other. But in our automation of descente infinie, the steps in the proof process depend on each other. The choice of the induction order, for instance, is delayed typically until all other subgoals except for order subgoals have been proved. Then, the automatic proof control can take into account all order constraints for choosing the induction order. Therefore, Step 3 is performed after the first two steps. Similarly the simplification process in Step 2 depends on those lemmas for which inductive case splits have been performed in Step 1. These lemmas may be applied during the simplification process as induction hypotheses—in addition to those supplied by the user. Because of these dependencies, the steps are not well suited for the integration in a single waterfall.

3.2.1 A Simple Waterfall

The application of a phase p to a goal G may be modeled with a boolean-valued function *apply-phase* as follows: *apply-phase*$(p, G, \mathcal{G})$ returns whether p has been applied to G successfully. For a successful application, the reference parameter $\mathcal{G}$ contains the new subgoals generated by p.

Function *simple-waterfall* contains the realization of a simple waterfall in pseudo code

1 **foreach** $p \in \mathit{phases}$ **do**
2   **if** $\mathit{apply\text{-}phase}(p, G, \mathcal{G}')$ **then**
3         $\mathcal{G} \leftarrow \emptyset$;
4   **foreach** $G' \in \mathcal{G}'$ **do**
5             $\mathcal{G} \leftarrow \mathcal{G} + \mathit{simple\text{-}waterfall}(G', \mathit{phases})$;
6   **return** $\mathcal{G}$;
7 **return** $\{G\}$;

Figure 3.2: **Function** *simple-waterfall*(G, *phases*)

(cf. Figure 3.2). It is called with the goal G to be simplified and the phases *phases* that define the waterfall. The function returns normalized goals w.r.t. the waterfall, i.e. goals that cannot be reduced with any phase of the waterfall anymore: If none of the phases can be applied successfully, a set consisting of the original goal is returned (Line 7). Otherwise, *simple-waterfall* is called recursively for each new goal and the resulting subgoals are returned (Line 3–6). We will enhance this simple waterfall with reuse mechanisms in Chapter 7.

The waterfall of `ACL2`, for instance, uses the following phases (cf. [KMM00]):

Simplify: This phase is "the heart of the theorem prover". For instance, it uses decision procedures for rational linear arithmetic and applies lemmas for rewriting. It normalizes formulas according to a propositional calculus and exploits type information.

Eliminate Destructors: In `ACL2`, destructor style is used for specifying defined functions. In this phase, destructor style is converted to constructor style which is more suitable for the remaining phases.

Use Equivalences: Negated equations in the goal clause can be used for replacing one side of the negated equation by the other side in another goal literal (cf. inference rule `const-rewrite` in Section 2.2.2.7). In this phase negated equations are used extensively, restricted by some heuristics to avoid infinite loops. Furthermore, this phase is not limited to negated equations but may also exploit other equivalence relations.

Generalize: This phase generalizes goals e.g. by replacing common terms with new variables.

Eliminate Irrelevance: This phase attempts to eliminate literals in the goals that seem to be irrelevant.

Induct: This phase constructs an induction scheme according to explicit induction (cf. Section 3.1.2).

3.2.2 A Flexible Waterfall

We use a waterfall for controlling the simplification process which is modeled with one single phase in ACL2. Therefore, we pose additional requirements on our waterfall model:

- It should be easy to integrate new phases into the waterfall. This property is exploited in Chapter 4 for integrating a decision procedure for linear arithmetic into the simplification process.
- It should be easy to configure the waterfall by exchanging phases: There is a great dependency between the order of the phases and the performance of the whole proof process. Little changes of the order may lead to significant improvements or losses in efficiency. But the dependency is fragile: What is good for one example may be bad for another example. We believe that—if there is a best order at all—it cannot be determined by theoretical considerations alone as the proof process is too complex. Therefore, we regard the search for a good order of the phases in the waterfall model as an engineering task: It has to be determined and validated by experiments.
- It should take into account that our waterfall—controlling the simplification process—works on a much more fine-grained level than a waterfall that implements the whole inductive proof process. Therefore, our waterfall should be highly configurable. The simple waterfall e.g. does not make any use of the information that a phase has been applied successfully. The handling of new subgoals always starts with the first phase. Instead, we want to be able to define a specific behavior as a response to a successful phase in an easy way.

To satisfy these requirements, we enhance the simple waterfall: To realize a more fine-grained control we divide the phases into *operations*. An operation focuses on one single literal in the goal. Furthermore, the implementation of our waterfall is split into three parts:

- a fixed *control tactic* which implements the recursive structure of the waterfall;
- the operations which constitute the basic elements of the phases; and
- a table-based configuration which establishes a connection between the first two parts and allows the user to fix the continuations of successful operations in a simple way.

3.2.2.1 The Control Tactic

The use of a single control tactic allows us to hide the complexity of the recursive structure within one fixed tactic which does not have to be modified when integrating new phases or operations into the waterfall. Furthermore, improvements of this control tactic have global effects on the whole waterfall.

Due to the complexity of the control tactic we do not provide any pseudo code for it in this thesis. In a nutshell, the recursive control structure is realized as a sequence of iterations. This allows us to choose the continuation of the waterfall in a flexible way according to the table-based configuration.

In comparison to the simple waterfall, the implementation of a phase is fixed in such a way that each literal is used as focus literal in succession. A focus literal is handled by calling those operations of the phase that are associated with the type of the literal. The *type* of a literal consists of the predicate symbol of the literal (=, `def`, <) and a flag whether the literal is negated.

If an operation is successful the table-based configuration may influence the behavior of the waterfall in two ways:

- Special actions may be determined for the new literals as well as for the (changed) focus literal in the new subgoals.
- It is determined whether the handling of new subgoals continues with the next phase or whether the waterfall is restarted from the beginning.

3.2.2.2 Operations

Operations contain the code to handle one focus literal in a goal. To enable code sharing, the operations may be parameterized. The parameters may be instantiated in the table-based configuration or manually when the user calls the control tactic. If no value is supplied a global default value will be used. Parameters are e.g. used for determining whether all lemmas should be considered for rewriting or subsumption or only the directly applicable lemmas.

We pose some simple requirements on the implementation of the operations so that the control tactic is able to perform its task:

- As for the phases of the simple waterfall, successful operations will not be deleted during the automatic proof attempt anymore. Therefore, operations are responsible for checking whether they are applicable. To achieve this, operations may recursively call the simplification process. But these calls should be restricted to perform complete proofs of certain subgoals. If an operation is not successful it is responsible for deleting all performed proof attempts.
- The control tactic has to determine the positions of the new literals and the focus literal in the new subgoals. As all the inference rules in QUODLIBET add new literals to the front of the goal clause this is quite simple. To support this task, operations may at most remove the focus literal in the goal clause. Furthermore, they have to inform the control tactic in which of the new subgoals the focus literal has been removed. More precisely, the operations define, for each new subgoal, a literal that should be used as focus literal (if there exists any).

3.2.2.3 Table-Based Configuration

The control may be influenced in a simple way by two tables: the operation and the phase table. The entries in the *operation table* refer to the code templates of the operations. Operations are combined to phases in the *phase table*. The parameters of the code templates may be instantiated in both tables preferring the local values given in the operation table.

An entry in the operation table consists of

(o1) a name to refer to the operation in the phase table;

(o2) a pointer to the parameterized code template of the operation;

(o3) an optional list of values to instantiate the parameterized operation;

(o4) flags whether the new literals and the (changed) focus literal of the new subgoals created by the operation should be handled with special actions;

(o5) an indicator for the continuation of the waterfall.

An entry in the phase table consists of

(p1) a name of the phase;

(p2) for each type of literal, a list of operations that are checked for applicability in succession;

(p3) an optional list of values to instantiate the operations of the phase in a uniform way;

(p4) an optional list of phases that should be applied to the new literals in the new subgoals created by a successful operation in this phase;

(p5) an optional list of phases that should be applied to the (changed) focus literals in the new subgoals created by a successful operation in this phase.

The waterfall handles all the phases in the phase table in succession until one of them can be applied successfully. The reactions to a successful operation are determined in the following way: First, the new literals in the new subgoals are handled with the phases defined in Item (p4) of the successful phase unless this is prevented with the corresponding flag in Item (o4) of the successful operation. Then, the focus literal is handled analogously with the phases in Item (p5) of the successful phase. At last, the waterfall is continued for the resulting subgoals as defined in Item (o5) of the successful operation. The waterfall may be restarted from the beginning immediately. Furthermore, we may choose to complete the phase for all the literals that were already present in the original goal, before we start the simplification process from the beginning. In this way, we can realize a fair handling of every literal in the goal clause. We may even choose to continue the simplification process with the next phase. This is sensible if we know that the first phases will fail anyway so that we do not have to check them once again. This is the case e.g. for a phase that only removes literals.

3.2.2.4 Phases of the Simplification Process of QuodLibet

As a starting point for the simplification process in this thesis, we use the waterfall as it is presented in [SS04]. The waterfall is divided into the following five[2] phases:

[2]As there are no means to prove negated definedness or negated order atoms apart from using them as complementary literals, they are not considered by the last three phases at all.

`prove-taut`: This phase proves simple tautologies that can be shown by a single application of one inference rule without using any lemmas. To be more precise, the inference rules for simple tautologies `compl-lit`, $\neq$`-taut` and $<$`-taut`, as well as the inference rules for decomposing atoms $=$`-decomp`, `def-decomp` and $<$`-decomp` are applied as long as they do not produce any new subgoals. Certainly, for each type of literal, only those inference rules are tried which are sensible for that type.

`remove-redundant`: During this phase all inference rules that remove redundant literals, i.e., `mult-lit`, $=$`-removal`, $<$`-removal`, $\neq$`-removal` and $\neg$`def-removal`, are attempted.

`reduce1`: This phase tries to apply non-permutative lemmas that are directly applicable to the goal clause. If the goal clause can be subsumed by one lemma, the goal is proved; otherwise, a normal form w.r.t. the directly applicable rewrite lemmas is computed testing for simple tautologies and redundant literals after each rewrite step. This phase has been split from `reduce2` to prefer directly applicable lemmas as they lead to easier proofs.

`reduce2`: During this phase a great effort is undertaken to prove the goal with the considered literal. Nearly all inference rules (except `const-rewrite`) that have not been applied during the first three phases are attempted. There are some special *macro inference steps*, i.e., sequences of inference steps that are linked together to simplify goals that contain a special pattern. They are used e.g. for removing constructor prefixes of definedness atoms and equational literals as well as for guessing intermediate weights for order atoms. Above all, the main focus of this phase is on applying subsumption and (possibly permutative) rewrite lemmas even if they are not directly applicable to the goal clause. During this phase some inference rules are tested, simplifying their derived subgoals by the whole simplification process recursively. But if some of the conditions, represented by the subgoals, cannot be established, the whole subtree is deleted again. This guarantees that at most one goal results from this phase (without counting goals with order atoms containing weight variables).

`cross-fertilize`: This phase affects only negated equational atoms. They are used for replacing in another literal the occurrence of one side of the negated equation with the other side by applying inference rule `const-rewrite`. To prevent infinite loops, the operations in this phase do not perform applications that undo former ones. Furthermore, the application of this phase is controlled by some heuristics to obtain "simpler" literals.

The waterfall is extended with new phases in Chapters 4 and 7. Heuristics for controlling the application of conditional lemmas in phases `reduce1` and `reduce2` are described in Chapter 6.

Chapter 4

An Even Closer Integration of Linear Arithmetic

In general, inductive validity is not even semi-decidable. However, it is decidable for some important theories such as linear arithmetic which occurs in many application domains. The corresponding specialized *decision procedures* are much more efficient than the heuristic search strategies which apply inference rules of a generic calculus. Furthermore, they relieve the user of the burden of having to speculate auxiliary lemmas of the theory. The properties expressed with the auxiliary lemmas are taken into account by the decision procedure automatically. To sum up, it is beneficial to integrate decision procedures for decidable subtheories into (inductive) theorem provers for two reasons: to gain efficiency and to broaden the scope of the (inductive) theorem prover, i.e. enhance its capabilities to prove theorems automatically. Because of their importance, decision procedures, the *combination* of decision procedures over *disjunctive* domains, and their *integration* into theorem provers have been studied for many decades. Research about the combination of decision procedures has been initiated by fundamental work in [NO79] and [Sho84]. Seminal work on the integration of decision procedures is presented in [BM88b] integrating a decision procedure for rational numbers based on Hodes [Hod71] (credited to Fourier in [KN94]) into their inductive theorem prover `NQTHM`. As we want to improve the automation of inductive theorem proving we focus on this last approach for integrating decision procedures.

Decision procedures decide the validity—or, its dual, unsatisfiability—of formulas over their underlying domain. If decision procedures are combined or integrated into theorem provers additional requirements arise: For efficiency reasons it is beneficial if the decision procedures do not provide only boolean answers but make explicit further consequences of the input formulas. These consequences may be exchanged with the other decision procedures or theorem provers. Therefore, not all the input formulas for a decision procedure need to be present at once but they may be added during the reasoning process. This favors *online* procedures [BGD03] which allow for adding and removing input formulas efficiently. For this, the decision procedure keeps track of its internal state. The internal state may be augmented with new formulas or reset to a former state with backtracking.

A decision procedure for theory T may be used for an extension of T by generalizing those terms whose top-level symbol is *alien* or *uninterpreted*, i.e. it is not covered by T. These alien subterms are replaced with new variables in a *variable abstraction step* in a *uniform* way, i.e. identical subterms are replaced with the same new variable. After having eliminated all alien subterms the formula belongs to T. Thus, the decision procedure is applicable to this generalized formula. If the generalized formula is unsatisfiable this holds true for the original formula as well. Otherwise, we have to "check" the solutions in the extended theory. For this, the result of the decision procedure may be adapted to the original problem by substituting the alien subterms for the corresponding new variables again. In practice, we do not have to introduce new variables at all. Instead, it suffices to slightly modify the decision procedures in such a way that they consider alien subterms as variables.

To integrate a decision procedure into an (inductive) theorem prover in a meaningful way, we have to make assumptions about the functioning of the decision procedure and the theorem prover. Following [BM88b], we assume that the decision procedure performs two major kinds of steps:

- A *variable elimination step* that combines two subformulas to a new formula eliminating at least one variable that occurs in both subformulas. The new formula is added to the state of the decision procedure.
- A check of *ground* instances for unsatisfiability (or validity).

Similar to ordered resolution, variable elimination steps need to be performed only for heaviest variables w.r.t. a fixed, total and wellfounded order: To get a ground instance, *all* variables have to be eliminated. Thus, the order of the variable elimination steps may be fixed without affecting satisfiability.

A theorem may be unsatisfiable in an extended theory but not in the theory of the decision procedure itself (cf. Example 4.2). Then, the unsatisfiability cannot be proved solely with the decision procedure. Instead, we have to use additional facts about the extension. These facts are usually given in the form of (conditional) lemmas. The conclusion of a lemma can be applied if the conditions of the lemma can be proved valid. These proofs may be performed by the decision procedure or the theorem prover. Thus, we get *mutual dependencies*.

The decision procedure and the theorem prover have to cooperate to find the right instance of an appropriate lemma. If the conclusion of a lemma is an inequality, it may be added to the state of the decision procedure to enable another variable elimination step of a heaviest variable. This mechanism is called *augmentation* in [BM88b]. It is suitable as long as the required lemmas are present but it does not provide any information if they are missing.

In this chapter, we present a new approach to incorporate a decision procedure *closely* into an (inductive) theorem prover. We strictly distinguish the logic part of the decision procedure from its control aspects: Each elementary step of the decision procedure is represented by a new *inference rule* providing the state of the decision procedure explicitly in the goals. Local properties of the inference rules guarantee the *soundness* of our approach. They may be applied automatically within tactics.

Our approach provides the following advantages: The fragmentation of the decision procedure into elementary steps—realized with inference rules—provides us with detailed *proof objects* which can be checked easily with an external proof checker. It also enables a *uniform* and *flexible* integration into our simplification process. This allows us to evaluate different integration schemes which are defined on a much more fine-grained level than in previous approaches. It also gives us the opportunity to implement different strategies: The integration into the simplification process uses the heuristics known from the literature to automate the decision procedure and to guide the augmentation mechanism. Furthermore, we have implemented a special purpose tactic that performs all possible variable elimination steps (but no other steps). We call this tactic only if the simplification process fails and we need more information to speculate an auxiliary lemma.

The new inference rules resulting from our close integration depend on the decision procedure and the theorem prover. Therefore, we have to fix these two parameters. Because of the importance of linear arithmetic for the verification of program and hardware designs, we integrate a decision procedure for this theory into our inductive theorem prover QuodLibet. As explained in [BM88b], the efficiency of the decision procedure itself is irrelevant when using it in an extended theory (cf. Section 4.4). Therefore, we choose the rather simple decision procedure based on Hodes which is well suited for the augmentation mechanism.

In Section 4.1, we present an overview of linear arithmetic, Hodes' decision procedure, and the augmentation mechanism. We illustrate the difficulties in speculating lemmas with an example. In Section 4.2, we present our new approach of a close integration. We enhance the logic of QuodLibet with new inference rules for the elementary steps of the decision procedure, and the waterfall with new phases that integrate the decision procedure into the automatic proof control of the theorem prover. Furthermore, we present a specialized tactic that supports the speculation of auxiliary lemmas. In Section 4.3, we validate our new approach with some case studies. We conclude this chapter with a survey of related work in Section 4.4.

4.1 Linear Arithmetic

By *linear arithmetic* we mean the first-order theory over predicate symbols $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ for order relations on numbers, and function symbols 0, s and $+$ for constant zero, unary successor function and binary addition.[1] We consider only the universally quantified fragment of linear arithmetic. Depending on the underlying domain, these theories are called *Presburger rational arithmetic* (PRA), *Presburger integer arithmetic* (PIA), and *Presburger natural arithmetic* (PNA) in [JB02]. We are interested in a semi-decision procedure for an *extended theory* of PNA containing additional predicate or function symbols. These symbols are *uninterpreted* for the decision procedure, but may be constrained by the theorem prover.

4.1.1 Hodes' Decision Procedure for Linear Arithmetic

Hodes' procedure can be used as a decision procedure for PRA and as a semi-decision procedure for PIA and PNA. It checks for unsatisfiability of a set of inequalities. The key idea is to "cross-multiply and add" [BM88b] two inequalities to eliminate a common variable

[1] For ease of use, we will also consider constant symbols for all numbers, $-$ for subtraction, and $\cdot$ for multiplication with constants ($n \cdot x$ abbreviates the sum that contains n times x).

in a variable elimination step. Variable elimination steps can be restricted to the heaviest variables in an inequality w.r.t. a fixed wellfounded order. In previous work, the inequalities are stored in an internal state of the decision procedure called *linear arithmetic data base* in [BM88b] or *constraint store* in [AR03, ARS02]. If an unsatisfiable (ground) inequality is derived, the original inequalities are unsatisfiable over rationals, integers, and naturals. Otherwise, if the set is closed under variable elimination steps, the original inequalities are satisfiable over the rationals but may be unsatisfiable over integers or naturals (cf. Section 4.2.3.4). We illustrate Hodes' procedure with a simple example.

Example 4.1 (derived from [BM88b]) We want to prove the validity of Formula (4.1) over the naturals. Therefore, we check its negation for unsatisfiability.

$$\forall K, L, \mathit{Max}, \mathit{Min}.(L \leq \mathit{Min} \wedge 0 < K \wedge \mathit{Min} \leq \mathit{Max} \rightarrow L < \mathit{Max} + K) \tag{4.1}$$

After normalizing the negation of (4.1), we get the conjunctively combined Inequalities (4.2) to (4.5). Note that we use the integral property of the naturals in Inequality (4.3): The difference of two unequal naturals is at least one.

$$L \leq \underline{\mathit{Min}} \tag{4.2}$$
$$1 \leq \underline{K} \tag{4.3}$$
$$\underline{\mathit{Min}} \leq \mathit{Max} \tag{4.4}$$
$$\underline{\mathit{Max}} + K \leq L \tag{4.5}$$

We restrict variable elimination steps using an alphabetic order on variable names. The heaviest variable of each inequality is underlined. Thus, we derive the following inequalities with an unsatisfiable ground Inequality (4.8):

$$L \leq \underline{\mathit{Max}} \qquad \text{from (4.2) and (4.4) eliminating } \mathit{Min} \tag{4.6}$$
$$\underline{K} \leq 0 \qquad \text{from (4.6) and (4.5) eliminating } \mathit{Max} \tag{4.7}$$
$$1 \leq 0 \qquad \text{from (4.3) and (4.7) eliminating } K \tag{4.8}$$

□

Example 4.1 falls into the decidable theory of PNA. But if we replace variables *Min* and *Max* with terms $\mathit{MIN}(A)$ and $\mathit{MAX}(A)$ as well as the third condition $\mathit{Min} \leq \mathit{Max}$ with $A \neq \mathit{nil}$ where A ranges over lists, then the formula is no longer valid in *pure* PNA, but contains uninterpreted function symbols. Therefore, we require a relationship between $\mathit{MIN}(A)$ and $\mathit{MAX}(A)$ which is valid only in the *extended* theory over lists. To apply this relationship, we use the *augmentation* mechanism described in [BM88b].

4.1.2 Augmentation

As already explained, additional facts of an extended theory are usually given in form of (conditional) lemmas. Following [KMM00], we call a lemma a *rewrite rule* if the conclusion of the lemma is an equation $s = t$; we call it a *linear rule* if the conclusion is an inequality $u \leq v$. The application of a rewrite rule replaces an instance of s with the same instance of t. In contrast to this, the application of a linear rule adds an instance of $u \leq v$ to the state of the decision procedure so that it can benefit from the new inequality. According to [BM88b], we *augment* the linear data base.

Example 4.2 (derived from [BM88b]) We want to prove the validity of Formula (4.9) over the naturals. We assume that Formula (4.10) is valid in the extended theory.

$$\forall A, K, L.(L \leq MIN(A) \wedge 0 < K \wedge A \neq nil \rightarrow L < MAX(A) + K) \tag{4.9}$$
$$\forall A.(A \neq nil \rightarrow MIN(A) \leq MAX(A)) \tag{4.10}$$

The decision procedure can make use of Inequalities (4.11) to (4.13) derived from the negation of (4.9). We assume that the decision procedure handles terms starting with uninterpreted function symbols just like variables over the naturals. Therefore, we omit explicit generalizations.

$$L \leq \underline{MIN(A)} \tag{4.11}$$
$$1 \leq \underline{K} \tag{4.12}$$
$$\underline{MAX(A)} + K \leq L \tag{4.13}$$

As the decision procedure tries to eliminate only the underlined heaviest terms in an inequality, it does not perform a single step (assuming the same order on the terms as in Example 4.1). But Lemma (4.10) may be applied as it contains additional information about the *heaviest*[2] term of (4.11). The condition of the lemma can be relieved because the same literal occurs in Formula (4.9). Therefore, we can augment the data base of the decision procedure with the conclusion of the lemma, namely $\underline{MIN(A)} \leq MAX(A)$. Then we can replay the proof from Example 4.1. □

Example 4.2 can be complicated further by replacing the condition $A \neq nil$ in Formula (4.9) with $length(A) > 0$, introducing another uninterpreted function symbol. Then, the condition of Lemma (4.10) is not directly present in Formula (4.9) but has to be relieved by recursively calling the simplifier of the theorem prover and the decision procedure. According to [HKM04], the integration of linear arithmetic into `ACL2` leads to four dependencies between the simplifier and the arithmetic package. Therefore, it is questionable whether the integration of linear arithmetic as a separate module is reasonable.

4.1.3 Lemma Speculation

The situation gets worse if the required lemmas are not present. To speculate rewrite rules automatically, successful approaches have been proposed e.g. in [KS03, GK03]. But for the automatic speculation of linear rules no general approach has been proposed yet. There exist approaches only for nonlinear arithmetic [HKM03, AR01]. As a linear rule contains an estimate, it is more difficult than for rewrite rules to speculate lemmas that are both—*valid* and *useful*. In our opinion, lemma speculation is a very creative task which has to be done by humans in most cases. But this task must be supported as far as possible. Therefore, we require an appropriate interaction scheme providing the human user with all the information needed. Previous approaches lack information for two reasons: First, they do not explicitly present the state of the decision procedure to the user. Instead, the information is hidden in the internal state of the decision procedure. Second, the decision procedure eliminates only heaviest terms.

[2]Typically, the augmentation mechanism is restricted in the same way as the variable elimination steps. Otherwise, it would not be guaranteed that the new inequality enabled further variable elimination steps.

Since we want to prove inductive validity instead of unsatisfiability with QuodLibet, we transform the procedure sketched in Section 4.1.1 using negation. The state of the decision procedure is represented directly within the goal clauses in form of new literals changed or added by the inference rules. The inference rules are flexible: They do not restrict variable elimination steps to heaviest terms but allow for the realization of different strategies. In the following example, we want to indicate how our new integration scheme supports the speculation of auxiliary lemmas required for the augmentation mechanism if these lemmas are not present.

Example 4.3 We consider Formula (4.9) from Example 4.2 (in clausal form) and want to derive Lemma (4.10). Figure 4.1 illustrates our derivation in form of a proof state tree (without displaying definedness subgoals and (negated) definedness atoms).

First, we try to prove the clause automatically using an extended waterfall which contains Hodes' decision procedure (cf. Section 4.2.4). This automatic proof attempt starts by normalizing all literals with inference rule `la-norm` (cf. Section 4.2.3). The literals (or terms) that are used by an inference rule are framed in Figure 4.1. Since the waterfall uses the heuristics to eliminate only heaviest terms, the proof attempt fails after the three normalization steps.

With a special purpose tactic for supporting the speculation of auxiliary lemmas (cf. Section 4.2.5), all variable elimination steps with inference rule $\leq$`-var-elim` are performed. This results in a last goal node which contains the needed auxiliary lemma as subformula, marked with frames in Figure 4.1. Therefore, we get a hypothesis for an auxiliary lemma. □

4.2 An Even Closer Integration

In this section, we describe our actual integration scheme. It consists of new derived inference rules for the elementary steps of the decision procedure and an automatic proof control realized with tactics. A preliminary version of the approach can be found in [Ron04].

To guarantee the soundness and safeness of our integration scheme, we start in Section 4.2.1 with a *base specification* $\mathtt{spec}_0$ that defines the required sorts and function symbols for the integration. The naturals provide one of the data models for $\mathtt{spec}_0$. In $\mathtt{spec}_0$, Hodes' decision procedure may be realized with tactics applying the generic inference rules from Section 2.2.2 and inductively valid lemmas w.r.t. $\mathtt{spec}_0$. But this realization is by far too inefficient. Therefore, we use the inductively valid lemmas only to prove the soundness and safeness of some derived inference rules. These inference rules make use of special *normal forms* of terms and literals over natural numbers which are particularly suitable for the steps of the decision procedure. These normal forms are defined in Section 4.2.2. In Section 4.2.3, we present the nine derived inference rules in detail. We integrate the new inference rules in the simplification process in Section 4.2.4, and present a special purpose tactic for supporting the speculation of auxiliary lemmas in Section 4.2.5.

$$\{\ \boxed{L < \texttt{+}(\texttt{MAX}(A), K)},\ \neg(L \leq \texttt{MIN}(A)),\ \neg(\texttt{0} < K),\ A = \texttt{nil}\ \}$$

`la-norm`

$$\{\ \texttt{+}(\texttt{1}, L) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \boxed{\neg(L \leq \texttt{MIN}(A))},\ \neg(\texttt{0} < K),\ A = \texttt{nil}\ \}$$

`la-norm`

$$\{\ \texttt{+}(\texttt{1}, L) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq L,\ \boxed{\neg(\texttt{0} < K)},\ A = \texttt{nil}\ \}$$

`la-norm`

$$\{\ \texttt{+}(\texttt{1}, \boxed{L}) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq \boxed{L},\ K \leq \texttt{0},\ A = \texttt{nil}\ \}$$

$\leq$`-var-elim`

$$\{\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq \texttt{+}(\boxed{K}, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, L) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq L,\ \boxed{K} \leq \texttt{0},\ A = \texttt{nil}\ \}$$

$\leq$`-var-elim`

$$\{\ \texttt{MIN}(A) \leq \texttt{MAX}(A),\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, L) \leq \texttt{+}(\boxed{K}, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq L,\ \boxed{K} \leq \texttt{0},\ A = \texttt{nil}\ \}$$

$\leq$`-var-elim`

$$\{\ L \leq \texttt{MAX}(A),\ \boxed{\texttt{MIN}(A) \leq \texttt{MAX}(A)},\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, L) \leq \texttt{+}(K, \texttt{MAX}(A)),\ \texttt{+}(\texttt{1}, \texttt{MIN}(A)) \leq L,\ K \leq \texttt{0},\ \boxed{A = \texttt{nil}}\ \}$$

Figure 4.1: Derivation of Lemma (4.10) from Formula (4.9)

4.2.1 The Base Specification $\texttt{spec}_0$

Our base specification $\texttt{spec}_0$ consists of two sorts

$\texttt{Bool}$ for boolean values with constructors

- $\texttt{true} : \to \texttt{Bool}$
- $\texttt{false} : \to \texttt{Bool}$

$\texttt{Nat}$ for natural numbers with constructors

- $\texttt{0} : \to \texttt{Nat}$
- $\texttt{s} : \texttt{Nat} \to \texttt{Nat}$

and the following defined operators:

- $\texttt{leq} : \texttt{Nat}, \texttt{Nat} \to \texttt{Bool}$
- $\texttt{+} : \texttt{Nat}, \texttt{Nat} \to \texttt{Nat}$
- $\texttt{-} : \texttt{Nat}, \texttt{Nat} \to \texttt{Nat}$
- $\texttt{*} : \texttt{Nat}, \texttt{Nat} \to \texttt{Nat}$

specified with the axioms presented in Figure 4.2 where m and n are constructor variables of sort $\texttt{Nat}$. If we want to prove theorems in an extended theory we may introduce new sorts with constructors, or defined operators with defining rules as usual, considering a constructor-consistent extension of $\texttt{spec}_0$ according to Definition 2.12.

In the following sections, we present a couple of inductively valid lemmas—such as the domain lemmas depicted in Figure 4.3—which may be used for implementing an extended

$$\{\ \texttt{leq}(\texttt{0}, n) = \texttt{true}\ \} \tag{4.14}$$
$$\{\ \texttt{leq}(\texttt{s}(m), \texttt{0}) = \texttt{false}\ \} \tag{4.15}$$
$$\{\ \texttt{leq}(\texttt{s}(m), \texttt{s}(n)) = \texttt{leq}(m, n)\ \} \tag{4.16}$$
$$\{\ \texttt{+}(m, \texttt{0}) = m\ \} \tag{4.17}$$
$$\{\ \texttt{+}(m, \texttt{s}(n)) = \texttt{s}(\texttt{+}(m, n))\ \} \tag{4.18}$$
$$\{\ \texttt{-}(m, \texttt{0}) = m\ \} \tag{4.19}$$
$$\{\ \texttt{-}(\texttt{0}, \texttt{s}(n)) = \texttt{0}\ \} \tag{4.20}$$
$$\{\ \texttt{-}(\texttt{s}(m), \texttt{s}(n)) = \texttt{-}(m, n)\ \} \tag{4.21}$$
$$\{\ \texttt{*}(m, \texttt{0}) = \texttt{0}\ \} \tag{4.22}$$
$$\{\ \texttt{*}(m, \texttt{s}(n)) = \texttt{+}(\texttt{*}(m, n), m)\ \} \tag{4.23}$$

Figure 4.2: Axioms of the Base Specification $\texttt{spec}_0$

$\{$ `def leq`(m, n) $\}$	(4.24)
$\{$ `def +`(m, n) $\}$	(4.25)
$\{$ `def -`(m, n) $\}$	(4.26)
$\{$ `def *`(m, n) $\}$	(4.27)

Figure 4.3: Inductively Valid Lemmas of $\mathtt{spec}_0$ for Proving Definedness Properties

version of Hodes' decision procedure. We employ these lemmas to prove the soundness and safeness properties of our new derived inference rules. To ease comprehension, we introduce the required lemmas just when needed.

Because of their importance for the decision procedure we represent constructor ground terms of sort `Nat` with the usual decimal constants, i.e. we write $i \in \mathbb{N}$ as an abbreviation for $\mathtt{s}^i(\mathtt{0})$. Thus, $\mathrm{top}(i) \in \{\mathtt{0}, \mathtt{s}\}$ for each constant $i \in \mathbb{N}$. Furthermore, we introduce $\leq$ as another predefined predicate symbol: we write $m \leq n$ instead of $\mathtt{leq}(m, n) = \mathtt{true}$ and $\neg(m \leq n)$ instead of $\mathtt{leq}(m, n) \neq \mathtt{true}$. This allows us to abandon sort `Bool` from the base specification. We call $m \leq n$ (resp. $\neg(m \leq n)$) a (negated) *inequality*. Furthermore, a (negated) equation, inequality or order atom with exactly one term of sort `Nat` on each side of the literal is called a *binary literal over sort* `Nat`.

4.2.2 Normal Forms

To support the variable elimination steps, we have to determine the number of occurrences of each term in a binary literal over sort `Nat`. Therefore, we define *polynomials* (cf. Section 4.2.2.1) and *normalized binary literals over sort* `Nat` (cf. Section 4.2.2.2). Note that, in other approaches, the representation of normal forms is hidden in the internal state of the decision procedure. In contrast to this, we represent normal forms explicitly in our close integration. This allows us to inform the user about the internal state of the decision procedure and, thus, to support the speculation of auxiliary lemmas. Normal forms are omnipresent in the specification of our derived inference rules. They have to obey only some simple requirements. Therefore, there are many ways to realize them appropriately. Finally, we fix one realization on a highly technical level to guarantee the *uniqueness* of the normal forms and, thus, of the subgoals generated with our derived inference rules.

4.2.2.1 Normal Forms for Terms of Sort `Nat`

The normal forms for terms of sort `Nat` are computed essentially by splitting the terms into single constituents, rearranging these single constituents, and combining identical ones. For this, we exploit fundamental properties of linear arithmetic such as the associativity and commutativity of addition and multiplication as well as the distributivity of multiplication over addition and subtraction. The required inductively valid lemmas are summarized in Figure 4.4.

In the following, we state this simple idea more precisely. We start with the definition of polynomials which contains the most important requirements on our normal forms. Then,

$$\{ \mathtt{+}(m, n) = \mathtt{+}(n, m) \} \tag{4.28}$$
$$\{ \mathtt{+}(\mathtt{+}(m_1, m_2), m_3) = \mathtt{+}(m_1, \mathtt{+}(m_2, m_3)) \} \tag{4.29}$$
$$\{ \mathtt{*}(m, n) = \mathtt{*}(n, m) \} \tag{4.30}$$
$$\{ \mathtt{*}(\mathtt{*}(m_1, m_2), m_3) = \mathtt{*}(m_1, \mathtt{*}(m_2, m_3)) \} \tag{4.31}$$
$$\{ \mathtt{*}(m, \mathtt{+}(n_1, n_2)) = \mathtt{+}(\mathtt{*}(m, n_1), \mathtt{*}(m, n_2)) \} \tag{4.32}$$
$$\{ \mathtt{*}(m, \mathtt{-}(n_1, n_2)) = \mathtt{-}(\mathtt{*}(m, n_1), \mathtt{*}(m, n_2)) \} \tag{4.33}$$
$$\{ \mathtt{-}(\mathtt{+}(m, n_1), \mathtt{+}(m, n_2)) = \mathtt{-}(n_1, n_2) \} \tag{4.34}$$

Figure 4.4: Inductively Valid Lemmas of $\mathtt{spec}_0$ for Normalizing Terms

we present an algorithm for computing specific normal forms. The latter representation provides the details for performing the splits, rearrangements and combinations of the constituents in our current implementation.

We assume $<_{\mathrm{Term}}$ to be a fixed, total, wellfounded order on terms $\mathrm{Term}(F, V)$ over an admissible specification $\mathtt{spec} = (\Sigma, C, R)$ (cf. Definition 2.5)—which is a constructor-consistent extension of $\mathtt{spec}_0$ according to Definition 2.12—with signature $\Sigma = (S, F, \alpha)$ and a set V of variable symbols.

Definition 4.4 (Polynomials/Addends/Coefficients/Multiplicands/Constants) P is a *polynomial* if $P \equiv c + \sum_{i=1}^{n} c_i t_i$ and for each $i, j \in \{1, \ldots, n\}$: $c, c_i \in \mathbb{N}$, $c_i \neq 0$, $t_i \in \mathrm{Term}(F, V)$, $\mathrm{top}(t_i) \notin \{\mathtt{0}, \mathtt{s}, \mathtt{+}\}$ and $t_i <_{\mathrm{Term}} t_j$ for $i < j$. $c_i t_i$ is called *addend*, c_i *coefficient*, t_i *multiplicand*, and c *constant* of the polynomial P. □

A polynomial can be easily represented as a term if we construct the sum with operators + and * with parenthesis associating to the right. Additionally, we eliminate factors with value 1 and constants with value 0. We identify polynomials with their term representation. Thus, we can use them e.g. in binary literals over sort Nat.

We present an algorithm *poly* that converts a term of sort Nat into a polynomial. As we are interested in merging as many common subterms as possible into one multiplicand, we exploit properties of the defined operators such as the commutativity of the multiplication and the distributivity of multiplication over the addition and subtraction. Therefore, we get polynomials that obey further restrictions. To formulate these restrictions we define *products*.

Definition 4.5 (Products / Factors) π is a *product* if $\pi \equiv \prod_{i=1}^{n} t_i$ and for each $i, j \in \{1, \ldots, n\}$: $t_i \in \mathrm{Term}(F, V)$, $\mathrm{top}(t_i) \notin \{\mathtt{0}, \mathtt{s}, \mathtt{*}\}$ and $t_i \leq_{\mathrm{Term}} t_j$ for $i < j$. t_i is called *factor* of the product π. □

Again, a product can be easily represented as a term using operator * with parenthesis associating to the right. We identify products with their term representation.

Algorithm *poly* obeys the following invariant (I) w.r.t. the generated polynomial $P \equiv c + \sum_{i=1}^{n} c_i t_i$:

- If $\mathrm{top}(t_i) = \mathtt{-}$, i.e. $t_i \equiv \mathtt{-}(u, v)$ for two terms u, v of sort Nat, then u and v are polynomials without any common multiplicand (obeying invariant (I) themselves). Furthermore, only one of the constants is unequal to 0.

- If $\text{top}(t_i) = \mathtt{*}$, then $t_i \equiv \prod_{j=1}^{m} u_j$ is a product and $\text{top}(u_j) \notin \{\mathtt{+}, \mathtt{-}\}$ for each $j \in \{1, \ldots, m\}$.

We present algorithm *poly* in a functional style. It depends on algorithms for adding, subtracting and multiplying two polynomials represented with symbols $+_p$, $-_p$ and $*_p$; adding and multiplying constants with polynomials represented with $+_c$ and $*_c$; and multiplying two multiplicands represented with $*_m$. We only sketch these algorithms in the following.

- $$poly(t) = \begin{cases} 0 & \text{if } t \equiv \mathtt{0} \\ 1 +_c poly(u) & \text{if } t \equiv \mathtt{s}(u) \\ poly(u) +_p poly(v) & \text{if } t \equiv \mathtt{+}(u, v) \\ poly(u) -_p poly(v) & \text{if } t \equiv \mathtt{-}(u, v) \\ poly(u) *_p poly(v) & \text{if } t \equiv \mathtt{*}(u, v) \\ t & \text{otherwise, i.e. if } \text{top}(t) \notin \{\mathtt{0}, \mathtt{s}, \mathtt{+}, \mathtt{-}, \mathtt{*}\} \end{cases}$$

  Thus, algorithm *poly* is recursively called for the arguments of operators `s`, `+`, `-`, and `*`. Then, the results are handled with the corresponding operations on polynomials. Note that a term starting with an uninterpreted function symbol is left unchanged although it may contain further interpreted function symbols in its subterms. This decision is motivated by the fact that recursive operators over the natural numbers have to be specified using `s` instead of `+` because of the confluence criterion of QUODLIBET (cf. Theorem 2.7). Therefore, axioms are less likely applicable to terms containing normalized subterms.

- The addition of a constant c to a polynomial $P \equiv d + \sum_{j=1}^{m} d_j v_j$ is represented with $c +_c P$. The constant of the new polynomial is $c + d$, the addends are left unchanged w.r.t. P.

- The addition of two polynomials $P_1 \equiv c + \sum_{i=1}^{n} c_i u_i$ and $P_2 \equiv d + \sum_{j=1}^{m} d_j v_j$ is represented with $P_1 +_p P_2$. The constant of the new polynomial is $c + d$, the sums are merged into one sum in ascending order of the multiplicands adding the coefficients of equal multiplicands.

  The addition of polynomials is associative and commutative. Therefore, we may omit parentheses and may swap polynomials which we want to add. We use the notion $\bigoplus_{i=1}^{k} P_i$ to add k polynomials $P_1, \ldots, P_k$.

- The subtraction of two polynomials $P_1 \equiv c + \sum_{i=1}^{n} c_i u_i$ and $P_2 \equiv d + \sum_{j=1}^{m} d_j v_j$ is represented with $P_1 -_p P_2$.

  First, for each polynomial P_i, $i \in \{1, 2\}$ a new polynomial P_i' is computed: The constant of P_1' is $c - d$ using a subtraction operator on natural numbers as specified with Axioms (4.19) to (4.21), i.e. the result is 0 if c is less or equal to d. The addends of P_1' are derived from P_1 by subtracting the coefficients of common multiplicands in P_2. Analogously, P_2' is derived from P_2 (and P_1). The resulting polynomials do not share any multiplicands, and at least one of the constants is 0.

If P_1' is equal to 0, i.e. the constant is 0 and there are no addends, then the result of the subtraction $P_1 -_p P_2$ is also 0. If P_2' is equal to 0, then the result is P_1'. Otherwise, the result is $P_1' - P_2'$. In each case, our invariant (I) is fulfilled.

- The multiplication of a constant c with a polynomial $P \equiv d + \sum_{j=1}^{m} d_j v_j$ is represented with $c *_c P$. The constant of the new polynomial is $c \cdot d$. For each multiplicand v_j, the coefficient is changed to $c \cdot d_j$.
- The multiplication of two polynomials $P_1 \equiv c + \sum_{i=1}^{n} c_i u_i$ and $P_2 \equiv d + \sum_{j=1}^{m} d_j v_j$ is represented with $P_1 *_p P_2$. The resulting polynomial is
$c \cdot d +_c c *_c \sum_{j=1}^{m} d_j v_j +_p d *_c \sum_{i=1}^{n} c_i u_i +_p \bigoplus_{i=1}^{n} \bigoplus_{j=1}^{m} (c_i \cdot d_j) *_c (u_i *_m v_j)$.
- The multiplication of two multiplicands u and v is represented with $u *_m v$. The computation results in a polynomial and depends on the top-level symbols of the multiplicands:
  - If $\text{top}(u) = \text{top}(v) = \texttt{-}$, i.e. $u \equiv \texttt{-}(u_1, u_2)$ and $v \equiv \texttt{-}(v_1, v_2)$ for polynomials u_1, u_2, v_1, v_2, then the result is $((u_1 *_p v_1) -_p (u_2 *_p v_1)) -_p ((u_1 *_p v_2) -_p (u_2 *_p v_2))$.
  - If $\text{top}(u) = \texttt{-}$, i.e. $u \equiv \texttt{-}(u_1, u_2)$ for polynomials u_1, u_2, but $\text{top}(v) \neq \texttt{-}$, then the result is $(u_1 *_p v) -_p (u_2 *_p v)$. Note that a multiplicand may be interpreted as polynomial with a single addend and coefficient 1.
  - If $\text{top}(v) = \texttt{-}$, i.e. $v \equiv \texttt{-}(v_1, v_2)$ for polynomials v_1, v_2, but $\text{top}(u) \neq \texttt{-}$, then the result is $(u *_p v_1) -_p (u *_p v_2)$.
  - Otherwise, we consider u and v as products. A term whose top-level symbol is unequal to `*` may be interpreted as product with a single factor. The factors of u and v are merged into a single new product with ascending order of the factors. The result is a polynomial that consists of one addend with coefficient 1 and the new product as multiplicand.

Note that algorithm *poly* obeys Definition 4.4 and invariant (I). A term can be transformed into (the term representation of) a polynomial using the axioms of Figure 4.2 and the inductively valid lemmas of Figure 4.4 as long as the term is defined. If necessary, the new derived inference rules introduce a case split to guarantee the definedness of the terms and literals they are applied to. They make use of the total definedness of operators `+`, `-`, and `*` (cf. Figure 4.3). This results in the following definedness conditions w.r.t. linear arithmetic.

Definition 4.6 (Definedness Conditions w.r.t. Linear Arithmetic)
The *minimal definedness positions w.r.t. linear arithmetic* $\text{MinDefPosLA}(t)$ of a term t are defined as the minimal positions in $\{p \in Pos(t) \mid \text{top}(t/p) \notin (C \cup \{\texttt{+}, \texttt{-}, \texttt{*}\} \cup V^C)\}$.
The *definedness conditions w.r.t. linear arithmetic* of a term t are defined as
$\text{DefCondLA}(t) = \{\neg \texttt{def}\ t/p \mid p \in \text{MinDefPosLA}(t)\}$.
The definedness conditions w.r.t. linear arithmetic are lifted to binary literals λ over sort `Nat` with top-level terms u and v by $\text{DefCondLA}(\lambda) = \text{DefCondLA}(u) \cup \text{DefCondLA}(v)$. □

Lemma 4.7 (Soundness Property of Algorithm *poly*) Let $\langle \Gamma; w \rangle$ be a goal, $m \in \mathbb{N}$, and $p \in Pos(\Gamma[m])$ such that $t \equiv \Gamma[m]/p$ is a term of sort `Nat`. If Γ contains $\text{DefCondLA}(t)$, then there exists a derivation using the inference rules of Section 2.2.2, the axioms of

Figure 4.2 and the lemmas of Figure 4.3 and 4.4 such that $\langle \Gamma[poly(t)]_{m.p}; w\rangle$ is the only open subgoal in the derivation. □

We illustrate algorithm *poly* with the following example.

Example 4.8 Let $\mathtt{f}, \mathtt{g}, \mathtt{h} : \mathtt{Nat} \to \mathtt{Nat}$ be uninterpreted function symbols.

$$\begin{aligned}
&poly(\texttt{*}(\texttt{-}(\mathtt{f}(m), \mathtt{h}(m)), \texttt{+}(\texttt{-}(\texttt{+}(\mathtt{f}(m), \mathtt{g}(m)), \mathtt{g}(m)), \mathtt{f}(m))))\\
&\quad = poly(\texttt{-}(\mathtt{f}(m), \mathtt{h}(m))) *_p poly(\texttt{+}(\texttt{-}(\texttt{+}(\mathtt{f}(m), \mathtt{g}(m)), \mathtt{g}(m)), \mathtt{f}(m))) && (1)\\
&\quad = \texttt{-}(\mathtt{f}(m), \mathtt{h}(m)) *_p ((\texttt{+}(\mathtt{f}(m), \mathtt{g}(m)) -_p \mathtt{g}(m)) +_p \mathtt{f}(m)) && (2)\\
&\quad = \texttt{-}(\mathtt{f}(m), \mathtt{h}(m)) *_p (\mathtt{f}(m) +_p \mathtt{f}(m)) && (3)\\
&\quad = \texttt{-}(\mathtt{f}(m), \mathtt{h}(m)) *_p \texttt{*}(\mathtt{2}, \mathtt{f}(m)) && (4)\\
&\quad = \mathtt{2} *_c (\texttt{-}(\mathtt{f}(m), \mathtt{h}(m)) *_m \mathtt{f}(m)) && (5)\\
&\quad = \mathtt{2} *_c ((\mathtt{f}(m) *_p \mathtt{f}(m)) -_p (\mathtt{h}(m) *_p \mathtt{f}(m))) && (6)\\
&\quad = \texttt{*}(\mathtt{2}, \texttt{-}(\texttt{*}(\mathtt{f}(m), \mathtt{f}(m)), \texttt{*}(\mathtt{f}(m), \mathtt{h}(m)))) && (7)
\end{aligned}$$

In Line (1), algorithm *poly* is recursively called for the arguments of operator `*`. The results will be combined with $*_p$. In Line (2), all recursive calls to *poly* have been performed. Furthermore, operations $-_p$ and $+_p$ have been applied for those terms that do not change. In Line (3), the common multiplicand $\mathtt{g}(m)$ is eliminated while subtracting two polynomials. In Line (4), common addends are combined while adding two polynomials. The multiplication of two polynomials is reduced to the multiplication of two multiplicands in Line (5). The result is presented in Line (6). Once again, the multiplication of the simplified polynomials is reduced to the multiplication of the multiplicands. During this multiplication the factors are sorted in ascending order leading to the result in Line (7).

The definedness conditions w.r.t. linear arithmetic are

$$\begin{aligned}
&\mathrm{DefCondLA}(\texttt{*}(\texttt{-}(\mathtt{f}(m), \mathtt{h}(m)), \texttt{+}(\texttt{-}(\texttt{+}(\mathtt{f}(m), \mathtt{g}(m)), \mathtt{g}(m)), \mathtt{f}(m))))\\
&\qquad\qquad = \{\neg\mathtt{def}\ \mathtt{f}(m), \neg\mathtt{def}\ \mathtt{g}(m), \neg\mathtt{def}\ \mathtt{h}(m)\}.
\end{aligned}$$

Note that $\neg\mathtt{def}\ \mathtt{g}(m)$ is one of the definedness conditions although `g` is not present in the resulting polynomial. □

4.2.2.2 Normal Forms for Binary Literals over Sort `Nat`

We lift the normalization from terms of sort `Nat` to binary literals over sort `Nat`. For our extended version of Hodes' decision procedure, inequalities and negated equations are particularly suited. They may be used for eliminating variables (cf. Sections 4.2.3.4 and 4.2.3.5). Our first aim is to reduce the literals to a uniform representation as far as possible. For this, we may

- replace order atoms over sort `Nat` with inequalities;[3]

[3]Note that the fixed wellfounded order of QuodLibet based on the length of constructor terms compares two natural numbers according to the usual less predicate on natural numbers. Therefore, we can use the decision procedure for order literals over sort `Nat` as well.

- eliminate top-level occurrences of operator `-` in multiplicands using Axioms (4.19) to (4.21);
- replace equations over sort `Nat` with two goals containing one new inequality each.

Whereas the first transformation results in a single new subgoal, the other two transformations generate at least two new subgoals. These case splits may be unnecessary: Assume, for instance, that we have two inequalities $P_1 \leq P_2$ and $P'_1 \leq P'_2$ such that P_2 and P'_1 contain a common multiplicand $\texttt{-}(x, y)$. Then we can perform a variable elimination step eliminating $\texttt{-}(x, y)$ without having to introduce a case split for operator `-`. Our second aim is to avoid case splits as far as possible. To cope with these two contradicting aims, we offer three different levels for normalizing binary literals over sort `Nat`. The first level performs those transformations that do not introduce case splits except for those required for the definedness conditions w.r.t. linear arithmetic. The second level, additionally, eliminates top-level occurrences of operator `-` in multiplicands. The third level, additionally, replaces equations with two inequalities to enable further variable elimination steps.

Each normalized literal obeys the following definition.

Definition 4.9 (Normalized Binary Literals over Sort `Nat`) A binary literal λ over sort `Nat` is *normalized* if $\lambda \equiv (P_1 \leq P_2)$, $\lambda \equiv (P_1 = P_2)$, or $\lambda \equiv (P_1 \neq P_2)$, where P_1 and P_2 are polynomials that do not share any multiplicand, one of the constants is equal to 0, and the set of coefficients of P_1 and P_2 is coprime, i.e. there does not exist a natural number greater than 1 that divides each coefficient of P_1 and P_2. □

The first normalization level performs only those transformations that are required for obeying Definition 4.9. Algorithm *norm1* returns for each binary literal λ over sort `Nat` a single literal—the normal form of λ w.r.t. the first normalization level. It performs the following steps, essentially using the lemmas of Figure 4.5:

(1) (Negated) order atoms and negated inequalities over sort `Nat` are replaced with inequalities using Lemmas (4.35) to (4.38):
This step transforms

$$\begin{array}{rcl} u < v & \text{into} & \texttt{+}(1, u) \leq v \\ \neg(u < v) & \text{into} & v \leq u \quad \text{and} \\ \neg(u \leq v) & \text{into} & \texttt{+}(1, v) \leq u. \end{array}$$

The result is a literal $t_1 \mathrel{\diamond} t_2$ where t_1 and t_2 are terms of sort `Nat` and $\diamond \in \{\leq, =, \neq\}$.

(2) The algorithm computes the polynomials $P_1 \equiv \mathit{poly}(t_1)$ and $P_2 \equiv \mathit{poly}(t_2)$ of t_1 and t_2.

(3) The algorithm eliminates common multiplicands and constants of P_1 and P_2 using Lemmas (4.39) to (4.41). Let P'_1 and P'_2 be the resulting polynomials. They do not share any multiplicand and one of the constants is equal to 0.

(4) In the last step, coprimality of all the coefficients of P'_1 and P'_2 is achieved by dividing each coefficient by the greatest common divisor g of all coefficients. If g divides the constants of P'_1 and P'_2, then *norm1* returns the corresponding literal where all coefficients and constants are divided by g using Lemmas (4.42) to (4.44).

Otherwise, i.e. if g does not divide the constant of P'_1 (resp. P'_2), the result is adapted depending on the predicate symbol $\diamond$:

$$\{\ \mathtt{leq}(m,n) \neq \mathtt{true},\ \mathtt{leq}(\mathtt{s}(n),m) \neq \mathtt{true}\ \} \tag{4.35}$$

$$\{\ \mathtt{leq}(\mathtt{s}(n),m) = \mathtt{true},\ \mathtt{leq}(m,n) = \mathtt{true}\ \} \tag{4.36}$$

$$\{\ m < n,\ \mathtt{leq}(\mathtt{s}(m),n) \neq \mathtt{true}\ \} \tag{4.37}$$

$$\{\ \mathtt{leq}(\mathtt{s}(m),n) = \mathtt{true},\ \neg(m < n)\ \} \tag{4.38}$$

$$\{\ \mathtt{leq}(\mathtt{+}(m,n_1),\mathtt{+}(m,n_2)) = \mathtt{leq}(n_1,n_2)\ \} \tag{4.39}$$

$$\{\ \mathtt{+}(m,n_1) = \mathtt{+}(m,n_2),\ n_1 \neq n_2\ \} \tag{4.40}$$

$$\{\ \mathtt{+}(m,n_1) \neq \mathtt{+}(m,n_2),\ n_1 = n_2\ \} \tag{4.41}$$

$$\{\ \mathtt{leq}(\mathtt{*}(m,n_1),\mathtt{*}(m,n_2)) = \mathtt{leq}(n_1,n_2),\ m = \mathtt{0}\ \} \tag{4.42}$$

$$\{\ \mathtt{*}(m,n_1) = \mathtt{*}(m,n_2),\ n_1 \neq n_2,\ m = \mathtt{0}\ \} \tag{4.43}$$

$$\{\ \mathtt{*}(m,n_1) \neq \mathtt{*}(m,n_2),\ n_1 = n_2,\ m = \mathtt{0}\ \} \tag{4.44}$$

$$\{\ \mathtt{leq}(\mathtt{+}(\mathtt{*}(m,n_1),k),\mathtt{*}(m,n_2)) = \mathtt{leq}(\mathtt{s}(n_1),n_2),\ \mathtt{leq}(m,k) = \mathtt{true},\ k = \mathtt{0}\ \} \tag{4.45}$$

$$\{\ \mathtt{leq}(\mathtt{*}(m,n_1),\mathtt{+}(\mathtt{*}(m,n_2),k)) = \mathtt{leq}(n_1,n_2),\ \mathtt{leq}(m,k) = \mathtt{true}\ \} \tag{4.46}$$

$$\{\ \mathtt{*}(m,n_1) \neq \mathtt{+}(\mathtt{*}(m,n_2),k),\ \mathtt{leq}(m,k) = \mathtt{true},\ k = \mathtt{0}\ \} \tag{4.47}$$

Figure 4.5: Inductively Valid Lemmas of $\mathtt{spec}_0$ for Normalizing Literals (1)

- If $\Bumpeq \equiv \leq$, then the result is rounded up for the constant of P_1' (cf. Lemma (4.45)) and rounded down for the constant of P_2' (cf. Lemma (4.46)).
- If $\Bumpeq \equiv =$, then the equation can be removed using inference rule `appl-lit-removal` and Lemma (4.47). In this case, *norm1* returns $0 = 1$ which is obviously redundant.
- If $\Bumpeq \equiv \neq$, then the negated equation can be proved with Lemma (4.47). In this case, *norm1* returns $0 \neq 1$ which is obviously inductively valid.

Note that the returned normalized literal obeys Definition 4.9. The new derived inference rules—except for those that normalize literals or terms—assume that the handled literals

$$\{\ \texttt{leq}(\texttt{+}(k_1, \texttt{-}(m,n)), k_2) = \texttt{leq}(\texttt{+}(k_1,m), \texttt{+}(k_2,n)),\ \texttt{leq}(n,m) \neq \texttt{true}\ \} \tag{4.48}$$

$$\{\ \texttt{leq}(k_1, \texttt{+}(k_2, \texttt{-}(m,n))) = \texttt{leq}(\texttt{+}(k_1,n), \texttt{+}(k_2,m)),\ \texttt{leq}(n,m) \neq \texttt{true}\ \} \tag{4.49}$$

$$\{\ \texttt{+}(k_1,m) = \texttt{+}(k_2,n),\ \texttt{+}(k_1, \texttt{-}(m,n)) \neq k_2,\ \texttt{leq}(n,m) \neq \texttt{true}\ \} \tag{4.50}$$

$$\{\ \texttt{+}(k_1, \texttt{-}(m,n)) = k_2,\ \texttt{+}(k_1,m) \neq \texttt{+}(k_2,n),\ \texttt{leq}(n,m) \neq \texttt{true}\ \} \tag{4.51}$$

$$\{\ \texttt{-}(m,n) = \texttt{0},\ \texttt{leq}(n,m) = \texttt{true}\ \} \tag{4.52}$$

Figure 4.6: Inductively Valid Lemmas of $\texttt{spec}_0$ for Normalizing Literals (2)

are normalized. This is checked using the applicability condition $norm1(\lambda) = \lambda$.

The second normalization level eliminates top-level occurrences of operator `-` in multiplicands. Note that the elimination of operator `-` in favor of operator `+` causes difficulties because we do not consider integers but only natural numbers. Therefore, algorithm *norm2* has to introduce case splits according to Axioms (4.19) to (4.21). Thus, the algorithm does not return a single literal but a set of pairs $\{(\Gamma_1; \lambda_1), \ldots, (\Gamma_k; \lambda_k)\}$ where Γ_i is a set of normalized inequalities and λ_i is a normalized literal w.r.t. the second normalization level for each $i \in \{1, \ldots, k\}$. The inequalities in Γ_i introduce a case split. They are called *linearization hypothesis* in [BM88b]. Literal λ_i is the normal form of the input literal λ w.r.t. the linearization hypothesis Γ_i. Basically, λ is equivalent to $(\bigvee \Gamma_1 \vee \lambda_1) \wedge \cdots \wedge \bigvee(\Gamma_k \vee \lambda_k)$ (cf. Lemma 4.10). Thus, we define the linearization hypotheses in such a way that they can be added to the subgoal clauses directly. For ease of comprehension, we may consider the results as implications using the conjunctively combined negated literals of the linearization hypotheses as premises. To reduce the complexity of the algorithm, we define an algorithm *norm2aux* which normalizes all the literals in a pair $(\Gamma; \lambda)$. Algorithms *norm2* and *norm2aux* are defined by mutual recursion. First, we present algorithm *norm2* which handles a single binary literal over sort `Nat`:

(1) The algorithm computes $\lambda' \equiv norm1(\lambda)$—the normal form of the input literal λ w.r.t. the first normalization level.

(2) Let $\lambda' \equiv P_1 \mathrel{\hat{\approx}} P_2$ with $P_1 \equiv c + \sum_{i=1}^{n} c_i u_i$, $P_2 \equiv d + \sum_{j=1}^{m} d_j v_j$ and $\mathrel{\hat{\approx}} \in \{\leq, =, \neq\}$.

If none of the multiplicands of P_1 and P_2 contains operator `-` as top-level operator, then $\{(\varnothing; \lambda')\}$ is returned.

Otherwise, let us assume that there exists an addend u_h in P_1 such that $u_h \equiv t_1 - t_2$ for $h \in \{1, \ldots, n\}$. Let P_1' be the polynomial that is derived from P_1 by eliminating the addend u_h, i.e. $P_1' \equiv c + \sum_{i=1}^{h-1} c_i u_i + \sum_{i=h+1}^{n} c_i u_i$. Then, the algorithm returns: $norm2aux((\{\neg(t_2 \leq t_1)\}; P_1' + c_h t_1 \mathrel{\hat{\approx}} P_2 + c_h t_2)) \cup norm2aux((\{t_2 \leq t_1\}; P_1' \mathrel{\hat{\approx}} P_2))$.

Otherwise, there exists an addend v_h in P_2 such that $v_h \equiv t_1 - t_2$ for $h \in \{1, \ldots, m\}$. This case is handled analogously to the previous case.

This step is justified because of the lemmas presented in Figure 4.6.

Algorithm *norm2aux* handles its input $(\{\lambda'_1, \ldots, \lambda'_m\}; \lambda)$ as follows:

(1) First, the algorithm computes the normal forms of all input literals w.r.t. the second normalization level. Let

$$\begin{aligned} &\mathit{norm2}(\lambda) = \{(\Gamma_1; \lambda_1), \ldots, (\Gamma_n; \lambda_n)\} \\ &\mathit{norm2}(\lambda'_k) = \{(\Gamma_1^{(k)}; \lambda_1^{(k)}), \ldots, (\Gamma_{n_k}^{(k)}; \lambda_{n_k}^{(k)})\} \qquad \text{for } k \in \{1, \ldots, m\}. \end{aligned}$$

(2) The algorithm returns the combination of all normal forms:

$$\begin{aligned} \{(\Gamma_i \cup \Gamma_{i_1}^{(1)} \cup \cdots \cup \Gamma_{i_m}^{(m)} \cup \{\lambda_{i_1}^{(1)}, \ldots, \lambda_{i_m}^{(m)}\}; \lambda_i) \mid \\ i \in \{1, \ldots, n\}, i_1 \in \{1, \ldots, n_1\}, \ldots, i_m \in \{1, \ldots, n_m\}\}. \end{aligned}$$

Note that the algorithms *norm2* and *norm2aux* terminate as the number of occurrences of operator `-` decreases in each recursive call to *norm2*. The returned literals are normalized and do not contain any occurrence of operator `-` as top-level symbol of a multiplicand.

The third normalization level is only applicable to equations. It replaces the equation with two inequalities. It is computed with algorithm *norm3* which transforms an input literal λ into a set of pairs $\{(\Gamma_1; \lambda_1), \ldots, (\Gamma_k; \lambda_k)\}$.

(1) First, the algorithm computes the normal form of the input literal λ w.r.t. the second normalization level. Let $\mathit{norm2}(\lambda) = \{(\Gamma_1; u_1 = v_1), \ldots, (\Gamma_n; u_n = v_n)\}$.

(2) The algorithm replaces each equation with two inequalities according to the lemmas in Figure 4.7. Thus, it returns $\{(\Gamma_i; u_i \leq v_i), (\Gamma_i; v_i \leq u_i) \mid i \in \{1, \ldots, n\}\}$.

To provide a uniform interface for the three normalization levels, we define algorithm *norm* that expects a binary literal λ over sort `Nat` and a normalization level as input. It returns a set of pairs $\{(\Gamma_1; \lambda_1), \ldots, (\Gamma_k; \lambda_k)\}$ that represents the normalization of λ w.r.t. the given normalization level:

$$\{\, m = n,\ \mathtt{leq}(m, n) \neq \mathtt{true},\ \mathtt{leq}(n, m) \neq \mathtt{true} \,\} \tag{4.53}$$

$$\{\, \mathtt{leq}(m, n) = \mathtt{true},\ m \neq n \,\} \tag{4.54}$$

$$\{\, \mathtt{leq}(m, n) = \mathtt{true},\ \mathtt{leq}(n, m) = \mathtt{true} \,\} \tag{4.55}$$

Figure 4.7: Inductively Valid Lemmas of $\mathtt{spec}_0$ for Normalizing Literals (3)

- $norm(\lambda, i) = \begin{cases} \{(\varnothing; norm1(\lambda))\} & \text{if } i = 1 \\ norm2(\lambda) & \text{if } i = 2 \\ norm3(\lambda) & \text{if } i = 3 \end{cases}$

Lemma 4.10 (Soundness Property of Algorithm *norm***)**
Let $\langle \Gamma; w \rangle$ be a goal, and $m \in \mathbb{N}$ such that $\Gamma[m]$ is a binary literal over sort `Nat`. Let $i \in \{1, 2, 3\}$, and $norm(\Gamma[m], i) \equiv \{(\Gamma_1; \lambda_1), \ldots, (\Gamma_k; \lambda_k)\}$. If Γ contains DefCondLA($\Gamma[m]$), then there exists a derivation using the inference rules of Section 2.2.2, the axioms of Figure 4.2 and the lemmas of Figures 4.3 to 4.7 such that

$$\langle \Gamma_1 \cup \Gamma[\lambda_1]_m; w \rangle, \ldots, \langle \Gamma_k \cup \Gamma[\lambda_k]_m; w \rangle$$

are the only open subgoals in the derivation. □

We illustrate the different normalization levels with the following example.

Example 4.11 (a) $norm1(\mathtt{*}(2, m) < \mathtt{+}(2, \mathtt{*}(2, n))) = m \leq n$:

First, $\mathtt{*}(2, m) < \mathtt{+}(2, \mathtt{*}(2, n))$ is transformed into $\mathtt{+}(1, \mathtt{*}(2, m)) \leq \mathtt{+}(2, \mathtt{*}(2, n))$. Then, the common parts of both sides are eliminated by subtracting one constant from the other and vice versa. This results in $\mathtt{*}(2, m) \leq \mathtt{+}(1, \mathtt{*}(2, n))$. Finally, the coefficients and constants are divided by the greatest common divisor of all coefficients, which is 2. As the constant appears on the right-hand side, the result of its division is rounded down.

(b) $norm(\mathtt{-}(m, n) = \mathtt{-}(m, k), 1) = \{\ (\varnothing; \mathtt{-}(m, n) = \mathtt{-}(m, k))\ \}$.

Thus, the equation is already normalized w.r.t. the first normalization level.

(c) $$\begin{aligned} norm(\mathtt{-}(m, n) = \mathtt{-}(m, k), 2) = \{\ & (\{\mathtt{+}(1, m) \leq n, \mathtt{+}(1, m) \leq k\};\ k = n), \\ & (\{\mathtt{+}(1, m) \leq n, k \leq m\};\ m = n), \\ & (\{n \leq m, \mathtt{+}(1, m) \leq k\};\ k = m), \\ & (\{n \leq m, k \leq m\};\ \mathtt{0} = \mathtt{0})\ \}: \end{aligned}$$

Since the equation is normalized w.r.t. the first normalization level, the computation of the second normalization level starts by performing a case split w.r.t. the first top-level occurrence of operator `-` in a multiplicand, namely $\mathtt{-}(m, n)$.

$$\begin{aligned} norm2(\mathtt{-}(m, n) = \mathtt{-}(m, k)) = {} & norm2aux(\ (\{\neg(n \leq m)\}; m = \mathtt{+}(\mathtt{-}(m, k), n))\ ) \cup \\ & norm2aux(\ (\{n \leq m\}; \mathtt{0} = \mathtt{-}(m, k))\ ) \end{aligned}$$

Thus, if we interpret the resulting pairs as implications, we get:

(1) if $n \leq m$, the subtrahend n is transferred to the right-hand side;

(2) if $m < n$, the difference $\mathtt{-}(m, n)$ is replaced with `0`.

Algorithm *norm2aux* calls *norm2* recursively for each literal resulting in

$$\begin{aligned} norm2(\neg(n \leq m)) &= (\varnothing; \mathtt{+}(1, m) \leq n) \\ norm2(m = \mathtt{+}(\mathtt{-}(m, k), n)) &= norm2aux(\ (\{\neg(k \leq m)\}; \mathtt{+}(m, k) = \mathtt{+}(n, m))\ ) \cup \\ &\ norm2aux(\ (\{k \leq m\}; m = n)\ ) \\ &= \{\ (\{\mathtt{+}(1, m) \leq k\}; k = n)\ \} \cup \{\ (\{k \leq m\}; m = n)\ \} \end{aligned}$$

and

$$\begin{aligned}
&norm2(n \le m) = (\varnothing; n \le m)\\
&norm2(\mathtt{0} = \mathtt{-}(m,k)) = norm2aux(\ (\{\neg(k \le m)\}; k = m)\ ) \cup\\
&\qquad\qquad\qquad\qquad\quad norm2aux(\ (\{k \le m\}; \mathtt{0} = \mathtt{0})\ )\\
&\qquad\qquad\qquad\qquad = \{\ (\{\mathtt{+}(\mathtt{1},m) \le k\}; k = m)\ \}\ \cup\ \{\ (\{k \le m\}; \mathtt{0} = \mathtt{0})\ \}
\end{aligned}$$

Then, the results are combined to the four normal forms above.

(d) $$\begin{aligned}
norm(\mathtt{-}(m,n) = \mathtt{-}(m,k), 3) = \{\ &(\{\mathtt{+}(\mathtt{1},m) \le n, \mathtt{+}(\mathtt{1},m) \le k\};\ k \le n),\\
&(\{\mathtt{+}(\mathtt{1},m) \le n, \mathtt{+}(\mathtt{1},m) \le k\};\ n \le k),\\
&(\{\mathtt{+}(\mathtt{1},m) \le n, k \le m\};\ m \le n),\\
&(\{\mathtt{+}(\mathtt{1},m) \le n, k \le m\};\ n \le m),\\
&(\{n \le m, \mathtt{+}(\mathtt{1},m) \le k\};\ k \le m),\\
&(\{n \le m, \mathtt{+}(\mathtt{1},m) \le k\};\ m \le k),\\
&(\{n \le m, k \le m\};\ \mathtt{0} \le \mathtt{0}),\\
&(\{n \le m, k \le m\};\ \mathtt{0} \le \mathtt{0})\ \}:
\end{aligned}$$

In the third normalization level, each equation resulting from the second normalization level is replaced with two inequalities. Note that equation $\mathtt{0} = \mathtt{0}$ may be proved with `=-decomp`. Thus, we will get an unnecessary case split if we do not simplify those literals that result from a previous normalization level.

□

4.2.3 Derived Inference Rules for Linear Arithmetic

In this section, we present the new inference rules which may be used for implementing Hodes' decision procedure for linear arithmetic. The inference rules are sound and safe as they are derived from the inference rules in Section 2.2.2.

Definition 4.12 (Derived Inference Rules) An inference rule of the form

<rule name> <parameters>
$$\frac{\langle \Gamma; w \rangle}{\langle \Gamma_1; w_1 \rangle \ldots \langle \Gamma_n; w_n \rangle}$$
if <applicability conditions>

is a *derived* inference rule w.r.t. $\mathtt{spec}_0$ if, for each instantiation that fulfills the applicability conditions, there exists a proof state tree with root goal node $\langle \Gamma; w \rangle$—generated with the inference rules of Section 2.2.2, the axioms of $\mathtt{spec}_0$ of Figure 4.2, and inductively valid lemmas w.r.t. $\mathtt{spec}_0$—such that $\langle \Gamma_1; w_1 \rangle \ldots \langle \Gamma_n; w_n \rangle$ are the only open goal nodes in the proof state tree. □

Lemma 4.13 (Soundness and Safeness of Derived Inference Rules)
Every derived inference rule is sound and safe. □

For each new inference rule, we present its formal definition, describe its task within the decision procedure and sketch its derivation with the inference rules of Section 2.2.2. Detailed derivations of the inference rules as well as further examples of their usage can be found in [Ron04]. From this, we get the following lemma.

Lemma 4.14 (Soundness and Safeness of the New Inference Rules)
All the inference rules for linear arithmetic presented in this section are derived inference rules. Thus, they are sound and safe. □

4.2.3.1 Derived Inference Rules for Normalizing Literals and Terms

In other approaches that integrate decision procedures into theorem provers there is no explicit correspondence to our inference rules for normalizing terms of and literals over sort `Nat`. Instead, these normal forms are hidden in the internal data structures of the decision procedure. Nevertheless, normal forms are essential for implementing decision procedures in an efficient way. The explicit representation of these normal forms in our close integration is a distinctive feature in comparison to other approaches. It allows for the representation of the internal state of the decision procedure which supports the user in speculating auxiliary lemmas without sacrificing efficiency.

Normal forms are omnipresent in all other derived inference rules. The inference rules require normalized literals as input and generate normalized literals as output. Because of their importance, normal forms have been presented extensively in Section 4.2.2. Therefore, we give only a short summary here: Inference rule `la-norm` normalizes binary literals over sort `Nat` (cf. Figure 4.8). The inference rule determines the multiplicands of the literal as well as the number of occurrences of each multiplicand as required for the variable elimination steps. Furthermore, this normal form facilitates the identification of tautological and redundant literals w.r.t. linear arithmetic. On the one hand, we want to simplify the literals as much as possible. On the other hand, we want to avoid unnecessary case splits. As a compromise, we provide the user with three different normalization levels. Whereas the first one avoids case splits as far as possible, the second one eliminates top-level occurrences of `-` introducing case splits w.r.t. the linearization hypotheses, and the third one replaces an equation with two inequalities. Because of the confluence criterion of QUODLIBET (cf. Theorem 2.7), constructor recursion of a defined operator over sort `Nat` may be specified only with constructors `0` and `s` but not with the defined operator `+`. Therefore, inference rule `la-norm` does not normalize any subterms of uninterpreted function symbols. This facilitates the application of the axioms of the defined operator. Instead, we provide an additional inference rule `la-term-norm` to initiate the normalization of a subterm explicitly.

`la-norm` is applicable if the considered literal is a binary literal over sort `Nat`. The second parameter of the inference rule determines the normalization level i. If $i = 3$, the considered literal has to be an equation. The inference rule creates, for each normal form computed with algorithm *norm*, one new subgoal replacing the considered literal with this normal form. Furthermore, the corresponding linearization hypothesis for the computation of the normal form is added to the subgoal. If the normal form differs from the considered literal in the original goal, then additional definedness subgoals are created for those definedness conditions w.r.t. linear arithmetic that are not present in the original goal.

`la-norm` m i

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle \ \ldots\ \langle \Lambda_k, \Gamma; w\rangle\ \langle \Gamma_1, \Lambda, \Gamma[\lambda_1]_m; w\rangle\ \ldots\ \langle \Gamma_n, \Lambda, \Gamma[\lambda_n]_m; w\rangle}$$

if there is an $n \in \mathbb{N}$ and there are literals $\lambda_1, \ldots, \lambda_n$ and clauses $\Gamma_1, \ldots, \Gamma_n$, and a minimal clause Θ such that

- $\Gamma[m]$ is a binary literal over sort `Nat`
- $i \in \{1, \ldots, 3\}$
- if $i = 3$, then $\Gamma[m]$ is an equation
- $norm(\Gamma[m], i) = \{(\Gamma_1; \lambda_1), \ldots, (\Gamma_n; \lambda_n)\}$
- if $n = 1$, $\Gamma_1 = \varnothing$, and $\lambda_1 = \Gamma[m]$, then $\Theta = \varnothing$; otherwise, Γ, Θ contains DefCondLA($\Gamma[m]$)
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

`la-term-norm` m p

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle \ \ldots\ \langle \Lambda_k, \Gamma; w\rangle\ \langle \Lambda, \Gamma[poly(\Gamma[m]/p)]_{m.p}; w\rangle}$$

if there is a minimal clause Θ such that

- $p \in Pos(\Gamma[m])$
- $\Gamma[m]/p$ is a term of sort `Nat`
- if $poly(\Gamma[m]/p) = \Gamma[m]/p$, then $\Theta = \varnothing$; otherwise, Γ, Θ contains DefCondLA($\Gamma[m]/p$)
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

Figure 4.8: Derived Inference Rules for Normalizing Literals and Terms

`la-term-norm` is applicable if the considered subterm is of sort `Nat`. The inference rule replaces the considered subterm with its polynomial as computed with algorithm *poly*. If the polynomial differs from the considered subterm in the original goal, then additional definedness subgoals are created for those definedness conditions w.r.t. linear arithmetic that are not present in the original goal.

For all derived inference rules, a formal derivation starts by possibly introducing a case split with inference rule `lit-add` for those definedness conditions w.r.t. linear arithmetic that are not present in the original goal. We do not present this part of the derivation for each inference rule explicitly but concentrate on the essential part that is different for each inference rule.

For `la-norm` and `la-term-norm`, the existence of the remaining derivation follows directly from the soundness properties of the algorithms *poly* and *norm* (cf. Lemmas 4.7 and 4.10).

$\leq$-`taut` m

$$\frac{\langle \Gamma; w \rangle}{\langle \Lambda_1, \Gamma; w \rangle \ \ldots \ \langle \Lambda_k, \Gamma; w \rangle}$$

if there is a polynomial P and a minimal clause Θ such that

- $\Gamma[m]$ is a normalized binary literal, i.e. $\mathit{norm1}(\Gamma[m]) = \Gamma[m]$
- $\Gamma[m] = (\mathtt{0} \leq P)$
- Γ, Θ contains $\mathrm{DefCondLA}(\Gamma[m])$
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

Figure 4.9: Derived Inference Rule for Proving Tautologies

4.2.3.2 Derived Inference Rule for Proving Tautologies

With inference rule $\leq$-`taut` (cf. Figure 4.9), we can identify inequalities that are tautological as required for Hodes' decision procedure.

$\leq$-`taut` is applicable if the considered literal is a normalized inequality of the form $\mathtt{0} \leq P$. Since we are concerned with inequalities over natural numbers only, each such inequality is inductively valid provided that the definedness conditions w.r.t. linear arithmetic are fulfilled. For those conditions that are not present in the goal, the inference rule generates corresponding definedness subgoals.

Note that we test arbitrary instances for tautologies with inference rule $\leq$-`taut`, not only ground ones.

A formal derivation of the inference rule just applies Axiom (4.14) (cf. Figure 4.2) for subsumption. We provide an inference rule for this derivation because we replace the defined operator `leq` with the predefined predicate symbol $\leq$. Therefore, we fix the semantics of $\leq$ with inference rules instead of axioms.

4.2.3.3 Derived Inference Rules for Removing Redundant Literals

We provide two inference rules for removing redundant inequalities (cf. Figure 4.10): Inference rule $\leq$-`removal` eliminates inequalities that are unsatisfiable by themselves. Inference rule $\leq$-`subs-removal` removes an inequality if it is more "restrictive" than another inequality in the goal. These inference rules are not required for the decision procedure but they clean up the goals making the presentation more concise. This helps the user to identify the important literals that are useful for speculating auxiliary lemmas.

$\leq$-`removal` is applicable if the considered literal is a normalized inequality of the form $P \leq \mathtt{0}$ and the constant in P is unequal to 0. The considered literal can be removed safely in the generated subgoal since it is unsatisfiable over natural numbers as long as P is defined. For this, a definedness subgoal is created for each definedness condition w.r.t. linear arithmetic that is not present in the original goal.

$\leq$-`removal` m

$$\frac{\langle \Gamma; w \rangle}{\langle \Lambda_1, \Gamma; w \rangle \ \ldots \ \langle \Lambda_k, \Gamma; w \rangle \ \langle \Lambda, \Gamma[-]_m; w \rangle}$$

if there is a polynomial $P \equiv c + \sum_{i=1}^{k} c_i t_i$ and a minimal clause Θ such that

- $\Gamma[m]$ is a normalized binary literal, i.e. $norm1(\Gamma[m]) = \Gamma[m]$
- $\Gamma[m] = (P \leq \mathtt{0})$
- $c > 0$
- Γ, Θ contains $\mathrm{DefCondLA}(\Gamma[m])$
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

$\leq$-`subs-removal` $m\ n$

$$\frac{\langle \Gamma; w \rangle}{\langle \Lambda_1, \Gamma; w \rangle \ \ldots \ \langle \Lambda_k, \Gamma; w \rangle \ \langle \Lambda, \Gamma[-]_n; w \rangle}$$

if there are polynomials $P_1 \equiv c + A_1, \quad P_2 \equiv d + A_2,$
$P'_1 \equiv c' + A_1, \quad P'_2 \equiv d' + A_2$
where $A_1 \equiv \sum_{i=1}^{k_1} c_i u_i$ and $A_2 \equiv \sum_{j=1}^{k_2} d_j v_j$,
and a minimal clause Θ such that

- $m \neq n$
- $\Gamma[m], \Gamma[n]$ are normalized literals, i.e. $norm1(\Gamma[m]) = \Gamma[m]$, $norm1(\Gamma[n]) = \Gamma[n]$
- $\Gamma[m] = (P_1 \leq P_2)$
- $\Gamma[n] = (P'_1 \leq P'_2)$
- $c + d' \leq d + c'$
- Γ, Θ contains $\mathrm{DefCondLA}(\Gamma[m]), \mathrm{DefCondLA}(\Gamma[n])$
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

Figure 4.10: Derived Inference Rules for Removing Redundant Literals

$\leq$-`subs-removal` is applicable if the two considered literals are inequalities with identical addends. Furthermore, the second one is more restrictive than the first one, i.e. the difference of the constants of the right-hand side and left-hand side of the first inequality is greater than or equal to that of the second inequality. In this case, the second inequality does not contain any additional information and can be removed safely provided that the definedness conditions w.r.t. linear arithmetic are fulfilled. This is guaranteed by generating corresponding definedness subgoals if needed.

With this inference rule, we may, e.g., remove $m \leq n$ using $m \leq \mathtt{+}(\mathtt{1}, n)$.

In the formal derivation of $\leq$-`removal`, we essentially apply `appl-lit-removal` with Axiom (4.15) (cf. Figure 4.2). For $\leq$-`subs-removal`, we apply `appl-lit-removal` with

Lemma (4.56) exploiting the condition $c + d' \leq d + c'$:[4]

$$\begin{aligned} \{\ & \texttt{leq}(\texttt{+}(m, c), \texttt{+}(n, d)) = \texttt{true}, \\ & \texttt{leq}(\texttt{+}(m, c'), \texttt{+}(n, d')) \neq \texttt{true}, \\ & \texttt{leq}(\texttt{+}(c, d'), \texttt{+}(d, c')) \neq \texttt{true}\ \} \end{aligned} \tag{4.56}$$

4.2.3.4 Derived Inference Rules for Eliminating Variables in Inequalities

One of the main steps of Hodes' decision procedure consists in performing variable elimination steps with inequalities: Two inequalities are cross-multiplied and added to eliminate a common multiplicand. In our approach, this step is performed with inference rule `≤-var-elim` which generates the new inequality in which at least one common multiplicand of the two considered input inequalities is eliminated (cf. Figure 4.11). Note that Hodes' decision procedure tests for unsatisfiability whereas our inference rules test for (inductive) validity. This transformation results in an adjustment $(k_1 + k_2 - 1)$ on the right-hand side of the new inequality (cf. Figure 4.11) caused by the negation of the inequalities.

If we apply Hodes' decision procedure to a set of inequalities over *pure* linear arithmetic, i.e. the inequalities do not contain any uninterpreted function symbols, then the following holds true: If we derive an unsatisfiable ground instance performing variable elimination steps, then the set of input inequalities is unsatisfiable over the rationals, the integers, and the naturals. Otherwise, if we close the set under variable elimination steps, the set of input inequalities is satisfiable over the rationals but may be unsatisfiable over integers or naturals. Therefore, in this simple form, Hodes' procedure is a decision procedure for rationals only. For each variable, we may derive an interval in which all constraints are satisfied. Whereas these intervals are guaranteed to be non-empty for rationals, there may be no integral solutions within the intervals. We may extend Hodes' procedure to get a decision procedure for integers and naturals. In short, we have to check the intervals for integral solutions. Extensions are presented e.g. in [KN94] and under the name of "Omega test" in [Pug92, BGD03].

In our approach, the same holds true for (inductive) validity: We have to provide another inference rule for checking the intervals of the rational solutions for integral ones to get a decision procedure for the naturals. This is done with `≤-case-split` by introducing a case split which allows us to check each integral value of the intervals. More precisely, the inference rule performs a case split in those cases where `≤-var-elim` generates an unsatisfiable ground instance $\hat{c} \leq 0$ with $\hat{c} \in \mathbb{N}$ and $\hat{c} > 0$. The case split introduces $\hat{c}$ new negated equations, one negated equation for each integral value of the interval. These negated equations may be used for removing one of the multiplicands with inference rules `la-const-rewrite` and `≠-var-elim` presented in Section 4.2.3.5.

`≤-var-elim` is applicable if the two considered literals are normalized inequalities which contain at least one common multiplicand. More precisely, there exists a common multiplicand in the right-hand side of the first inequality and in the left-hand side of

[4] More precisely, we introduce a case split with `lit-add` according to this literal. The subgoal that contains this literal can be proved with `la-norm` and `≤-taut`. The subgoal that contains its negation is applicable for `appl-lit-removal` with Lemma (4.56). Finally, the negation can be removed with `la-norm` and `≤-removal`.

$\leq$`-var-elim` $m\ n\ p$

$$\frac{\langle \Gamma; w \rangle}{\langle \Lambda_1, \Gamma; w \rangle\ \ldots\ \langle \Lambda_k, \Gamma; w \rangle\ \langle \lambda, \Lambda, \Gamma; w \rangle}$$

if there are polynomials P_1, P_2, P_1', P_2', a literal λ and a minimal clause Θ such that

- $\Gamma[m], \Gamma[n]$ are normalized literals, i.e. $\mathit{norm1}(\Gamma[m]) = \Gamma[m]$, $\mathit{norm1}(\Gamma[n]) = \Gamma[n]$
- $\Gamma[m] = (P_1 \leq P_2)$
- $\Gamma[n] = (P_1' \leq P_2')$
- $p \in \mathit{Pos}(\Gamma[m])$ is a position that refers to a multiplicand t in P_2 with coefficient $k_2 > 0$
- P_1' contains multiplicand t with coefficient $k_1 > 0$
- $\lambda = \mathit{norm1}(k_1 * P_1 + k_2 * P_1' \ \leq\ k_1 * P_2 + k_2 * P_2' + (k_1 + k_2 - 1))$
- Γ, Θ contains DefCondLA($\Gamma[m]$)
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

$\leq$`-case-split` $m\ n$

$$\frac{\langle \Gamma; w \rangle}{\langle \Lambda_1, \Gamma; w \rangle\ \ldots\ \langle \Lambda_k, \Gamma; w \rangle\ \langle \lambda_1, \Lambda, \Gamma; w \rangle\ \ldots\ \langle \lambda_{(c+c')-(d+d'+1)}, \Lambda, \Gamma; w \rangle}$$

if there are polynomials $P_1 \equiv c + A_1$, $P_2 \equiv d + A_2$, $P_1' \equiv c' + A_2$, $P_2' \equiv d' + A_1$, where $A_1 \equiv \sum_{i=1}^{k_1} c_i u_i$ and $A_2 \equiv \sum_{j=1}^{k_2} d_j v_j$,
and a minimal clause Θ such that

- $\Gamma[m], \Gamma[n]$ are normalized literals, i.e. $\mathit{norm1}(\Gamma[m]) = \Gamma[m]$, $\mathit{norm1}(\Gamma[n]) = \Gamma[n]$
- $\Gamma[m] = (P_1 \leq P_2)$
- $\Gamma[n] = (P_1' \leq P_2')$
- $c + c' > d + d' + 1$
- $\lambda_i = \mathit{norm1}(P_1 \ \neq\ i + P_2)$ for $i \in \{1, \ldots, (c + c') - (d + d' + 1)\}$
- Γ, Θ contains DefCondLA($\Gamma[m]$)
- $\Lambda_1, \ldots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

Figure 4.11: Derived Inference Rules for Eliminating Variables in Inequalities

the second inequality. The common multiplicand to be eliminated is identified with the third parameter which is the position of the multiplicand in the first inequality.

Beside the usual definedness subgoals to fulfill the definedness conditions w.r.t. linear arithmetic, the inference rule generates one subgoal that contains a new inequality in which the considered multiplicand is eliminated.

The soundness of the inference rule essentially depends on the monotonicity of the addition in both arguments. Thus, we can add the left-hand sides and the right-hand sides of two inequalities, respectively, to derive a new inequality. The normalization of the new inequality eliminates a common multiplicand if the number of occurrences of the multiplicand is the same on both sides of the inequality. This is guaranteed if we add k_1 occurrences of the first inequality and k_2 occurrences of the second inequality where k_1 (resp. k_2) is the coefficient of the multiplicand in the second (resp. first) inequality.

The adjustment of the right-hand side of the new inequality with (k_1+k_2-1) is justified as we actually exploit the monotonicity of the addition w.r.t. the *less* predicate on natural numbers: Since we use the input inequalities as premises they have to be negated (cf. Example 4.15 and Lemma (4.57) below).

$\leq$`-case-split` is applicable if the two considered literals are normalized inequalities. Furthermore, the addends of the left-hand side of one inequality are identical to the addends of the right-hand side of the other inequality and vice versa. The sum $c+c'$ of the constants of the left-hand side of both inequalities is at least greater by 2 than the sum $d+d'$ of the constants of the right-hand side of both inequalities.

Beside the usual definedness subgoals to fulfill the definedness conditions w.r.t. linear arithmetic, the inference rule generates $(c+c')-(d+d'+1)$ new subgoals. For each $i \in \{1, \ldots, (c+c')-(d+d'+1)\}$, the ith new subgoal contains an additional negated equation—the normalization of $P_1 \neq i+P_2$ where P_1 (resp. P_2) is the left-hand (resp. right-hand) side of the first considered inequality.

With the new negated equations, we explicitly consider those cases that prevent a proof with inference rules $\leq$`-var-elim` and $\leq$`-taut`: We consider the original goal under the assumption that $P_1 = i+P_2$ for $i \in \{1, \ldots, (c+c')-(d+d'+1)\}$.

Note that we do not restrict inference rule $\leq$`-var-elim` to heaviest terms since this is not important for its soundness. Instead, our inference rule is more general. Its automatic application is restricted by heuristics implemented with tactics.

Example 4.15 (a) We may apply $\leq$`-var-elim` to a goal containing literals

$$\texttt{+}(k, \texttt{*}(\texttt{2}, n)) \leq \texttt{+}(\texttt{1}, \texttt{*}(\texttt{3}, m)) \qquad \text{and}$$
$$\texttt{+}(\texttt{*}(\texttt{2}, k), m) \leq \texttt{*}(\texttt{3}, n).$$

If we want to eliminate variable n, then three occurrences of the first inequality are added to two occurrences of the second inequality. After having combined common multiplicands on each side, the resulting inequality is

$$\texttt{+}(\texttt{*}(\texttt{7}, k), \texttt{+}(\texttt{*}(\texttt{2}, m), \texttt{*}(\texttt{6}, n))) \leq \texttt{+}(\texttt{7}, \texttt{+}(\texttt{*}(\texttt{9}, m), \texttt{*}(\texttt{6}, n))).$$

Note that the constant contains the adjustment $(3+2-1)=4$. The elimination of the common multiplicands on both sides results in

$$\texttt{*}(\texttt{7}, k) \leq \texttt{+}(\texttt{7}, \texttt{*}(\texttt{7}, m)).$$

Intuitively, we may derive this inequality as follows: First note that $\lambda_1 \vee \lambda_2$ is logically equivalent to $\lambda_1 \vee \lambda_2 \vee \lambda_3$ provided that $(\neg\lambda_1 \wedge \neg\lambda_2)$ entails $\neg\lambda_3$. Therefore, we may add a literal λ_3 to a goal clause if its negation follows from the negation of literals λ_1 and λ_2 of the clause. If the two input inequalities do not hold, then

$$\texttt{+}(k, \texttt{*}(\texttt{2}, n)) > \texttt{+}(\texttt{1}, \texttt{*}(\texttt{3}, m)) \qquad \text{and}$$
$$\texttt{+}(\texttt{*}(\texttt{2}, k), m) > \texttt{*}(\texttt{3}, n).$$

From this, we get

$$\texttt{+}(\texttt{2}, \texttt{*}(\texttt{3}, m)) \leq \texttt{+}(k, \texttt{*}(\texttt{2}, n)) \qquad \text{and}$$
$$\texttt{+}(\texttt{1}, \texttt{*}(\texttt{3}, n)) \leq \texttt{+}(\texttt{*}(\texttt{2}, k), m).$$

Multiplying the first inequality with $\texttt{3}$ and the second one with $\texttt{2}$, we get

$$\texttt{+}(\texttt{6}, \texttt{*}(\texttt{9}, m)) \leq \texttt{+}(\texttt{*}(\texttt{3}, k), \texttt{*}(\texttt{6}, n)) \qquad \text{and}$$
$$\texttt{+}(\texttt{2}, \texttt{*}(\texttt{6}, n)) \leq \texttt{+}(\texttt{*}(\texttt{4}, k), \texttt{*}(\texttt{2}, m)).$$

The addition of the two inequalities results in

$$\texttt{+}(\texttt{8}, \texttt{*}(\texttt{7}, m)) \leq \texttt{*}(\texttt{7}, k).$$

Then, its negation may be added to the goal, namely,

$$\texttt{*}(\texttt{7}, k) \leq \texttt{+}(\texttt{7}, \texttt{*}(\texttt{7}, m)).$$

In fact, the inference rule divides all coefficients and constants by the greatest common divisor of all coefficients, which is 7. Thus, we get the new inequality

$$k \leq \texttt{+}(\texttt{1}, m)$$

which is added to the front of the new subgoal.

We may also eliminate variable m by combining one occurrence of the first inequality with three occurrences of the second inequality. This results in the new literal

$$k \leq n.$$

Note that we do not have to perform both variable elimination steps to get a complete decision procedure. Instead, it suffices to eliminate heaviest variables in both inequalities—in this case variable n.

(b) The inequalities

$$\texttt{+}(\texttt{1}, \texttt{max}(m, n)) \leq m \qquad \text{and}$$
$$\texttt{+}(\texttt{1}, m) \leq \texttt{max}(m, n)$$

contain the same addends on opposite sides. Since the difference of the constants on the left-hand sides and on the right-hand sides is 2, we may apply $\leq$`-case-split`. The application results in one new negated equation, namely

$$\texttt{max}(m, n) \neq m$$

which is added to the front of the new subgoal.

Intuitively, if the two input inequalities do not hold, then

$$m \leq \mathtt{max}(m,n) \qquad \text{and}$$
$$\mathtt{max}(m,n) \leq m$$

Therefore,

$$\mathtt{max}(m,n) = m$$

and we may add its negation to the goal.

In general, the inference rule may result in huge case splits. We restrict its automatic application in such a way that only a single new subgoal is created (except for definedness subgoals). Nevertheless, sometimes it may be beneficial to apply the inference rule manually even if it introduces more than one new subgoal.

□

In the formal derivation of $\leq$`-var-elim`, we essentially apply inference rule `lemma-subs` with Lemma (4.57):

$$\begin{aligned}\{\ & \mathtt{leq}(\mathtt{+}(\mathtt{*}(k_1,m_1),\mathtt{*}(k_2,n_1)),\mathtt{+}(\mathtt{*}(k_1,m_2),\mathtt{+}(\mathtt{*}(k_2,n_2),\mathtt{-}(\mathtt{+}(k_1,k_2),\mathtt{1})))) \neq \mathtt{true},\\ & \mathtt{leq}(m_1,m_2) = \mathtt{true},\\ & \mathtt{leq}(n_1,n_2) = \mathtt{true},\\ & k_1 = \mathtt{0},\\ & k_2 = \mathtt{0}\ \}\end{aligned} \tag{4.57}$$

For $\leq$`-subs-removal`, we introduce a case split with inference rule `lit-add` and literals $P_1 = i + P_2$ for $i \in \{1, \ldots, (c+c') - (d+d'+1)\}$ resulting in the following clauses for the new subgoals:

$P_1 \neq 1 + P_2,\ \Lambda, \Gamma$

$P_1 \neq 2 + P_2,\ P_1 = 1 + P_2,\ \Lambda, \Gamma$

...

$P_1 \neq (c+c') - (d+d'+1) + P_2,\ P_1 = (c+c') - (d+d'+2) + P_2,\ \ldots,\ P_1 = 1 + P_2,\ \Lambda, \Gamma$

$P_1 = (c+c') - (d+d'+1) + P_2,\ P_1 = (c+c') - (d+d'+2) + P_2,\ \ldots,\ P_1 = 1 + P_2,\ \Lambda, \Gamma$

Except for the last subgoal, the additional equations may be handled by applying inference rule `const-rewrite` with the first negated equation to replace P_1 with the right-hand side of the negated equation. After normalization, the resulting equations are unsatisfiable ground instances which may be removed with $=$`-removal`. Therefore, we derive the required subgoals.

In the last subgoal, we may derive $P_1 \leq (c+c') - (d+d'+1) + P_2$ using Lemma (4.58) with the first inequality $P_1 \leq P_2$ and each of the new equations in succession:

$$\begin{aligned}\{\ & \mathtt{leq}(m,n) = \mathtt{true},\\ & \mathtt{leq}(m,\mathtt{+}(\mathtt{1},n)) \neq \mathtt{true},\\ & m = \mathtt{+}(\mathtt{1},n)\ \}\end{aligned} \tag{4.58}$$

We may handle the resulting inequality by applying $\leq$`-var-elim` with the second inequality $P'_1 \leq P'_2$. This results in inequality $\mathtt{0} \leq \mathtt{0}$ which may be proved with $\leq$`-taut` finishing the derivation of $\leq$`-case-split`.

4.2.3.5 Derived Inference Rules for Eliminating Variables with Negated Equations

Theoretically, we could replace each negated equation with two inequalities and apply $\leq$`-var-elim` to perform variable elimination steps with inequalities. But it is much more efficient to exploit the equality information of negated equations directly as proposed in an extended version of Hodes' decision procedure in [KN94] based on [Knu81]. We can use negated equations for constantly rewriting the other literals in the goal clause as provided by inference rule `const-rewrite` (cf. Section 2.2.2.7). In doing so, we may remove one of the multiplicands of the negated equation in the other literals of the goal. This is possible as long as the coefficient of the multiplicand in the negated equation divides the coefficient of the multiplicand in the other literal. To achieve this property, we may "solve" the negated equation, i.e. we may derive a set of negated equations which represents a general integral solution of the negated equation using Euclid's algorithm. Each negated equation of the general integral solution contains at least one addend with coefficient 1. Therefore, the negated equation may be used for eliminating the corresponding multiplicand in all other literals of the goal.

In a nutshell, the general integral solution is computed as follows (cf. Example 4.16): If one of the coefficients in the considered (normalized) negated equation is equal to 1, we are done. Otherwise, we choose one addend A in the negated equation with minimal coefficient n. We divide each coefficient and the constant of the negated equation by n, rounding up (resp. down) for coefficients and constants that appear on the same (resp. opposite) side of the negated equation w.r.t. A. In doing so, we get a new negated equation where the coefficient of A is equal to 1. To compensate for the rounding error, we add a new addend consisting of a new constructor variable with coefficient 1 to the opposite side of the new negated equation w.r.t. A. The new negated equation allows us to eliminate the multiplicand of A in all other literals of the goal, in particular, in the negated equation considered first. For this negated equation, the elimination has the following consequences: The addend A is replaced with a new addend consisting of the new constructor variable and the same coefficient n; the constant and the coefficients of the other multiplicands are reduced to the reminder of the corresponding division by n. We may repeat this computation of introducing new negated equations and eliminating one of the multiplicands in the original negated equation until one of the coefficients in this negated equation is equal to 1. In doing so, we add a set of "solved" negated equations to the goal and "solve" the original negated equation itself. The general solution replaces one negated equation with $N + 1$ negated equations containing N new variables. As each negated equation in the general solution contains an addend with coefficient 1, we may replace $N + 1$ multiplicands in favor of N new variables. Therefore, the number of different multiplicands in the other literals of the goal is reduced by one.

For the implementation of this approach, we provide two inference rules that perform the elementary steps (cf. Figure 4.12): The first inference rule actually performs an elimination step. It can be applied for those multiplicands in a negated equation whose coefficient is 1 to eliminate this multiplicand in another normalized binary literal. It is called `la-const-rewrite` because of its affinity with `const-rewrite`. Inference rule $\neq$`-var-elim` supports the applicability of `la-const-rewrite`: It introduces a new negated equation which contains a new variable. The new negated equation may be used for reducing the `la-const-rewrite` $m\ n\ p$

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle \ \dots\ \langle \Lambda_k, \Gamma; w\rangle\ \langle \Lambda, \Gamma[\lambda]_n; w\rangle}$$

if there are polynomials P_1, P_2, P'_1, P'_2, a literal λ and a minimal clause Θ such that

- $\Gamma[m], \Gamma[n]$ are normalized literals, i.e. $\mathit{norm1}(\Gamma[m]) = \Gamma[m]$, $\mathit{norm1}(\Gamma[n]) = \Gamma[n]$
- $\Gamma[m] = (P_1 \neq P_2)$
- $\Gamma[n] = (P'_1 \mathrel{\dot{\sim}} P'_2)$ with $\dot{\sim} \in \{\leq, =, \neq\}$
- $p \in \mathit{Pos}(\Gamma[m])$ is a position that refers to a multiplicand t with coefficient 1
- P'_1 or P'_2 contains multiplicand t with coefficient $c > 0$
- if multiplicand t occurs in P_1 and P'_1 or in P_2 and P'_2, then

  $\lambda = \mathit{norm1}(P'_1 + c * P_2 \mathrel{\dot{\sim}} P'_2 + c * P_1)$;

  otherwise, multiplicand t occurs in P_1 and P'_2 or in P_2 and P'_1, and

  $\lambda = \mathit{norm1}(P'_1 + c * P_1 \mathrel{\dot{\sim}} P'_2 + c * P_2)$
- Γ, Θ contains $\mathrm{DefCondLA}(\Gamma[m])$
- $\Lambda_1, \dots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

$\neq$`-var-elim` $m\ p\ x$

$$\frac{\langle \Gamma; w\rangle}{\langle \Lambda_1, \Gamma; w\rangle \ \dots\ \langle \Lambda_k, \Gamma; w\rangle\ \langle \lambda, \Lambda, \Gamma; w\rangle}$$

if there are polynomials $P_1 \equiv c + \sum_{i=1}^{k_1} c_i u_i, \qquad P_2 \equiv d + \sum_{j=1}^{k_2} d_j v_j$

P'_1, P'_2, a literal λ and a minimal clause Θ such that

- $\Gamma[m]$ is a normalized binary literal, i.e. $\mathit{norm1}(\Gamma[m]) = \Gamma[m]$
- x is a constructor variable of sort `Nat` with $x \notin V(\Gamma, w)$
- $\Gamma[m] = (P_1 \mathrel{\dot{\neq}} P_2)$ such that $p \in \mathit{Pos}(\Gamma[m])$ is a position that refers to a multiplicand t in P_1 with coefficient $n > 1$
- $n \leq c_i$ and $n \leq d_j$ for each $i \in \{1, \dots, k_1\}$ and for each $j \in \{1, \dots, k_2\}$
- $P'_1 = \left\lceil \frac{c}{n} \right\rceil + \sum_{i=1}^{k_1} \left\lceil \frac{c_i}{n} \right\rceil u_i \qquad$ and $\qquad P'_2 = \left\lfloor \frac{d}{n} \right\rfloor + \sum_{j=1}^{k_2} \left\lfloor \frac{d_j}{n} \right\rfloor v_j$
- $\lambda = \mathit{norm1}(x + P'_2 \ \neq\ P'_1)$
- Γ, Θ contains $\mathrm{DefCondLA}(\Gamma[m])$
- $\Lambda_1, \dots, \Lambda_k, \Lambda$ is the case analysis resulting from Θ.

Figure 4.12: Derived Inference Rules for Eliminating Variables in Negated Equations

coefficients of the original negated equation by applying `la-const-rewrite` according to the description above.

`la-const-rewrite` is applicable if the two considered literals are normalized. Furthermore, the first literal is a negated equation. The third parameter is a position that refers to a multiplicand with coefficient 1 in this negated equation. The multiplicand is also present in the second literal with coefficient $c > 0$.

Beside the usual definedness subgoals to fulfill the definedness conditions w.r.t. linear arithmetic, the inference rule generates one additional subgoal in which the second considered literal is replaced with a new literal of the same type in which the considered multiplicand is eliminated. For this, c occurrences of the first negated equation are added to the second literal in such a way that the considered multiplicand occurs c times on both sides of the literal. The multiplicand is then removed via normalization.

$\neq$`-var-elim` is applicable if the considered literal is a normalized negated equation. Furthermore, the second parameter is a position that refers to a multiplicand in the negated equation with a smallest coefficient $n > 1$ in the negated equation. The third parameter is a new constructor variable of sort `Nat`.

Beside the usual definedness subgoals to fulfill the definedness conditions w.r.t. linear arithmetic, the inference rule generates one new subgoal that contains a new negated equation. The new negated equation is derived from the old one by dividing all coefficients and constants by n. The result is rounded up (resp. down) for those coefficients and constants that belong to the same (resp. opposite) side of the negated equation as the considered multiplicand. To compensate for the rounding, the new constructor variable is added to the opposite side.

Note that the coefficient of the considered multiplicand is 1 in the new negated equation. Therefore, we may apply `la-const-rewrite` to eliminate the multiplicand from the original negated equation reducing the coefficients of the other multiplicands.

Example 4.16 To illustrate the usage of $\neq$`-var-elim` and `la-const-rewrite` we consider a goal containing the negated equation

$$\texttt{*}(\texttt{6}, m_1) \neq \texttt{+}(\texttt{2}, \texttt{+}(\texttt{*}(\texttt{7}, m_2), \texttt{*}(\texttt{5}, m_3))).$$

Since none of the coefficients is 1, in general, the negated equation cannot be exploited to eliminate one of the variables in the other binary literals over sort `Nat`. But we may apply $\neq$`-var-elim` using variable m_3 to reduce the other coefficients in the literal. The application generates a new subgoal with

$$\texttt{+}(m_1, n) \neq \texttt{+}(\texttt{1}, \texttt{+}(\texttt{*}(\texttt{2}, m_2), m_3))$$

added to the front of the subgoal. In the negated equation, n is a new constructor variable of sort `Nat`. All coefficients and constants are divided by `5`—the coefficient of m_3. We add the new variable to the left-hand side—the opposite side w.r.t. m_3. Therefore, the results are rounded down (resp. up) on the left-hand (resp. right-hand) side. In the new negated equation, at least two variables have coefficient 1, namely, the new variable n and variable m_3 for which the inference rule is applied. We may use this negated equation for eliminating

m_3 in all other literals of the goal with inference rule `la-const-rewrite`. The application to the original negated equation adds five occurrences of the new negated equation (with swapped sides) to the original one. The resulting normalized negated equation

$$\texttt{+}(\texttt{3}, \texttt{+}(m_1, \texttt{*}(\texttt{3}, m_2))) \neq \texttt{*}(\texttt{5}, n)$$

replaces the original negated equation. The coefficient of the new variable n is equal to the coefficient of the eliminated variable m_3 but all other coefficients are reduced. In this case, the coefficient of variable m_1 is 1. Therefore, we may apply `la-const-rewrite` to eliminate m_1 in all other literals of the goal. The negated equation introduced by $\neq$`-var-elim`, for instance, may be replaced with

$$\texttt{*}(\texttt{6}, n) \neq \texttt{+}(\texttt{4}, \texttt{+}(\texttt{*}(\texttt{5}, m_2), m_3)).$$

Altogether, we get two negated equations which allow for the elimination of the old variables m_1 and m_3 in favor of the new variable n. □

In the formal derivation of `la-const-rewrite`, we essentially apply Lemmas (4.39) to (4.41) (cf. Figure 4.5) to add c occurrences of the left-hand side of the first literal to both sides of the second literal. Then, we apply `const-rewrite` to replace the left-hand side of the first literal with its right-hand side in the second literal in such a way that both sides contain c occurrences of the considered multiplicand. Finally, the new literal is normalized with `la-norm`.

Essentially, we start the formal derivation of $\neq$`-var-elim` by applying `lit-add` with literal $\neg$`def` $(P'_1 - P'_2)$ resulting in two new subgoals with clauses

$$\texttt{def}\ (P'_1 - P'_2),\ \Lambda,\ \Gamma$$
$$\neg\texttt{def}\ (P'_1 - P'_2),\ \Lambda,\ \Gamma$$

The definedness atom of the first subgoal can be proved with the lemmas of Figure 4.3 and the definedness conditions w.r.t. linear arithmetic. To the negated definedness atom in the second subgoal, we apply `ctr-var-add` with the new constructor variable x resulting in the negated equation $x \neq P'_1 - P'_2$. With Axiom (4.17) (cf. Figure 4.2) and Lemmas (4.50) and (4.51) (cf. Figure 4.6) this negated equation may be replaced with $x + P'_2 \neq P'_1$ possibly generating an additional condition subgoal with a new literal $P'_2 \leq P'_1$. Because of the rounding in the definition of P'_1 and P'_2, there exist two terms t_1 and t_2 such that $P'_1 = P_1 + t_1$ and $P'_2 = P_2 - t_2$. We can perform this restructuring with the lemmas of Figure 4.4. Thus, we may replace $P'_2 \leq P'_1$ with $P_2 - t_2 \leq P_1 + t_1$. To this literal, we may apply Lemma (4.59) generating another condition subgoal with a new inequality $P_2 \leq P_1$.

$$\{\ \texttt{leq}(\texttt{-}(m, k_1), \texttt{+}(n, k_2)) = \texttt{true},\ \texttt{leq}(m, n) \neq \texttt{true}\ \} \tag{4.59}$$

But this may be proved with Lemma (4.54) (cf. Figure 4.7) using the negated equation $P_1 \neq P_2$.

4.2.3.6 Concluding Remarks about the Derived Inference Rules

In Sections 4.2.3.1 to 4.2.3.5, we have presented our new derived inference rules for the integration of an extended version of Hodes' decision procedure for linear arithmetic into our inductive theorem prover QuodLibet. For the proof of the soundness and safeness of the inference rules, we have sketched formal derivations w.r.t. $\texttt{spec}_0$ using the axioms of Figure 4.2 and the lemmas of Figures 4.3 to 4.7 as well as Lemmas (4.56) to (4.59). Except for Lemma (4.38), we have formally proved the inductive validity of these lemmas with QuodLibet using the inference rules in Section 2.2.2. Lemma (4.38) can be proved as well if we assume the inductive validity of the following lemmas:

$$\begin{aligned}&\{\ \neg(m < n),\\ &\ \ m \neq n\ \}\end{aligned} \tag{4.60}$$

$$\begin{aligned}&\{\ \neg(m < n),\\ &\ \ \neg(n < m)\ \}\end{aligned} \tag{4.61}$$

These lemmas hold true in all data models because $<$ is a wellfounded order for each data model.[5]

4.2.4 Enhancing the Simplification Process

For the integration of Hodes' decision procedure for linear arithmetic into the simplification process of the inductive theorem prover QuodLibet, we have to enhance its waterfall (cf. Section 3.2.2). Essentially, we have to

- identify and implement appropriate operations; and
- adapt and extend the table-based configuration.

4.2.4.1 Identification of Operations

Most of the operations are determined directly by the decision procedure and the corresponding inference rules. Therefore, we get operations that

- normalize binary literals over sort `Nat` w.r.t. the different normalization levels;
- normalize terms of sort `Nat`;
- prove simple tautologies for inequalities;
- remove redundant inequalities;
- perform variable elimination steps using negated equations; and
- perform variable elimination steps for two inequalities.

Since we are concerned with a decision procedure on natural numbers, we may exploit the fact that $\texttt{0} \leq t$ holds true for each defined term t of sort `Nat`. Therefore, we get another operation that introduces a case split with inference rule `lit-add` using literal $\neg(\texttt{0} \leq t)$ for

[5] Nevertheless, we cannot prove these lemmas formally with the inference rules of QuodLibet because they do not provide means for proving negated order atoms except for inference rule `compl-lit` which does not help.

a term t that is the heaviest multiplicand in an inequality and appears on its right-hand side. Whereas $\texttt{0} \leq t$ can be proved with $\leq$`-taut`, $\mathit{norm1}(\neg(\texttt{0} \leq t)) = 1 +_c t \leq \texttt{0}$ allows us to eliminate multiplicand t in the inequality with $\leq$`-var-elim`.

To handle extended theories, we provide operations that implement the augmentation mechanism. They apply linear rules, i.e. lemmas whose head literal is an inequality, for subsumption even if the instantiated head literal is not present in the goal. It is only required that the head literal allows for at least one variable elimination step.

Furthermore, we have to introduce new operations and adapt some of the old operations because our normal forms introduce an alternative representation for terms of sort `Nat`. They favor operator `+` in combination with constants for natural numbers over operator `s` which in turn must be used for specifying axioms of defined operators. This complicates the application of axioms and lemmas. Theoretically, it would be beneficial to perform rewriting modulo the theory of linear arithmetic including associativity and commutativity of operators `+` and `*`. We may achieve this with the present inference rules by introducing a case split using `lit-add` with $t \neq \hat{t}$ where $\hat{t}$ is an alternative representation of t w.r.t. linear arithmetic. Then, the equation $t = \hat{t}$ is provable with inference rules `la-norm` and $=$`-decomp`. The negated equation $t \neq \hat{t}$ may be used for replacing t with $\hat{t}$ with inference rule `const-rewrite`. This allows us to use the alternative representation, e.g., for performing a rewrite step. The additional literal may be removed with `la-norm` and $\neq$`-removal`.

In practice, rewriting modulo linear arithmetic "can get very expensive" [KS96b]. Therefore, we have not implemented the corresponding matching operation. Instead, we consider only the equivalence of $(c + \sum_{i=1}^{n} c_i t_i)$ and $\texttt{s}((c-1) + \sum_{i=1}^{n} c_i t_i)$ if $c > 0$. This allows us to use axioms and lemmas for rewriting even if they are specified with operator `s`. We also exploit this equivalence to prove goals containing a negated equation $c + \sum_{i=1}^{n} c_i t_i \neq \texttt{0}$ with $c > 0$ using $\neq$`-taut`.

4.2.4.2 Implementation of Operations

For the implementation of the operations, we essentially employ heuristics known from the literature—in particular [BM88b]—to restrict proof search. Therefore, we do not present any technical details of the implementation in this thesis but give only a short summary of the restrictions.

- During the simplification process we restrict variable elimination steps with inference rule $\leq$`-var-elim` to heaviest multiplicands in both inequalities. For this purpose we use a fixed total simplification order on terms that at first depends on the number of variables, the term length and the name of the top-level operator or variable. If all these values are equal for both terms, the first unequal argument terms are considered recursively. This order is inspired by [BM88a].

- During variable elimination steps with negated equations, the applications of inference rule `la-const-rewrite` are restricted to one fixed multiplicand with coefficient 1 to prevent infinite loops.

- Case splits with $\neg(\texttt{0} \leq t)$ are performed only if this enables another variable elimination step (with $\leq$`-var-elim` or $\leq$`-case-split`) or the augmentation mechanism.

- The augmentation mechanism is essentially restricted in three ways:

  (1) Its applicability is checked only for those multiplicands that have been activated by the user. Multiplicands may be activated only if their top-level operators are uninterpreted.

  (2) *Extra* variables—i.e. those variables in the lemma that are not bound by matching the activated multiplicand to a multiplicand in the focus literal—have to be bound by matching other multiplicands or literals of the lemma to corresponding multiplicands or literals in the goal.

  (3) According to the restrictions of variable elimination steps for inequalities, the augmentation mechanism applies only those lemmas that allow for the elimination of a heaviest multiplicand.

4.2.4.3 The New Table-Based Configuration

Our new proof control integrates the operations for linear arithmetic into the former waterfall (cf. Section 3.2.2) on a fine-grained level. It combines the phases in such a way that the cheapest phases that promise the highest profit are handled first. Therefore, we interleave previous phases with new phases. Altogether, we get the following phases:

`prove-taut`: This phase is retained from the former waterfall. It is enhanced with the operation that proves simple tautologies for inequalities. Furthermore, the operation for negated equations over sort `Nat` takes into account the alternative term representation of polynomials.

`remove-redundant`: The former phase is supplemented with the new operation that removes redundant inequalities.

`def1`: Since all the inference rules for linear arithmetic may generate new definedness subgoals, these are handled first. Therefore, we provide this new phase which contains those operations of the former phase `reduce1` that handle definedness atoms.

`def2`: Analogously, this phase contains those operations of the former phase `reduce2` that handle definedness atoms.

`la-norm1`: This phase normalizes binary literals over sort `Nat` w.r.t. the first normalization level.

`la-norm2`: This phase normalizes binary literals over sort `Nat` w.r.t. the second normalization level. Note that the separation of the normalization levels in different phases may be beneficial although our waterfall contains both phases in succession: If the first normalization phase is applied successfully the waterfall will be restarted. Thus, tautological literals are proved and redundant literals are removed before the second normalization phase is considered.

`reduce1-nonaltrep`: In this phase, we perform the operations of the former phase `reduce1` without considering alternative term representations. This allows the user to speed up computations by providing appropriate auxiliary lemmas.

`la-var-elim1`: In this phase, we perform variable elimination steps with negated equations over sort `Nat` that do not contain any uninterpreted function symbols. Thus, we prefer those negated equations that are "pure" w.r.t. linear arithmetic.

`la-var-elim2`: In this phase, we perform variable elimination steps with negated equations over sort `Nat` that contain at least one uninterpreted function symbol.

`la-var-elim3`: In this phase, we perform variable elimination steps with inequalities.

`reduce1`: This phase is retained from the former waterfall. It is enhanced by considering alternative term representations.

`reduce2`: This phase is also improved in comparison to the former waterfall by considering alternative term representations.

`la-term-norm`: In this phase, subterms of sort `Nat` are normalized.

`subsume-left-leq`: In this phase, the augmentation mechanism is applied for multiplicands that occur on the left-hand side of an inequality in the goal. Since an inequality is tautological if its left-hand side is equal to `0`, we prefer those augmentation steps that eliminate multiplicands on the left-hand side.

`subsume-right-leq`: In this phase, the augmentation mechanism is applied for multiplicands that occur on the right-hand side of an inequality in the goal.

`subsume-negeq`: In this phase, the augmentation mechanism is applied for multiplicands that occur in a negated equation.

`la-add-multiplicand`: In this phase, we call the operation that performs a case split with $\neg(\mathtt{0} \leq t)$ to eliminate a multiplicand on the right-hand side of an inequality.

`la-norm3`: This phase normalizes binary literals over sort `Nat` w.r.t. the third normalization level.

`cross-fertilize`: This phase is retained from the former waterfall.

4.2.5 A Tactic for Supporting Speculation of Auxiliary Lemmas

Independently from our simplification process, we have implemented a special purpose tactic to facilitate the speculation of auxiliary lemmas for the augmentation mechanism. This tactic performs all variable elimination steps possible, without considering the heuristics to eliminate only heaviest terms. To guarantee termination, the tactic performs all variable elimination steps for a term only once. It starts with the heaviest multiplicand w.r.t. $<_{\mathrm{Term}}$ that occurs in an inequality. Finally, we purge the new clause with the phases `prove-taut` and `remove-redundant` of the waterfall. The effects of this tactic on the manual speculation of auxiliary lemmas are considered in Section 4.3.3.

4.3 Case Studies

In this section, we validate our new approach in three ways: Within QuodLibet, we compare the old proof control before the integration with the new proof control after the integration (cf. 4.3.1). The comparison of our new approach with other approaches such as the *extended proof method* `EPM` [JB02] and *constraint contextual rewriting* `CCR(X)` [AR03] demonstrates that our approach is quite competitive (cf. 4.3.2). Furthermore, we illustrate the additional benefits of our new approach w.r.t. the speculation of auxiliary lemmas (cf. 4.3.3).

4.3.1 Effects on Proof Control Within QuodLibet

In this section, we evaluate the effects of the integration of linear arithmetic into the simplification process of QuodLibet with some case studies. The results are summarized in Table 4.1. Certainly, examples are suited for the integration to a greater or lesser extent. We expect most benefits for examples that contain many "linear dependencies", i.e. literals that combine operators `+` and $\leq$. Furthermore, the evaluation of arithmetic expressions containing large constants is supported well because normalization is performed in a single step instead of dozens of lemma applications. For examples that do not contain any of the interpreted operators of linear arithmetic, we may, at best, expect unchanged runtimes. In practice, the runtimes usually slightly increase because the phases for linear arithmetic are checked for applicability unsuccessfully.

For the comparison we use the following case studies which are listed in ascending order w.r.t. their suitability for linear arithmetic:

`sortalgos`: This example contains a collection of sorting algorithms such as bubblesort, insertionsort, mergesort and quicksort. More details on the specifications can be found in [Kai02]. Although the algorithms are specified on natural numbers using $\leq$ for sorting, they are not well suited for linear arithmetic as there are only a few occurrences of operator `+` in the specifications.

`gcd`: In this case study, we prove that the greatest common divisor of two natural numbers is associative, commutative and idempotent. For the proofs, we exploit dependencies between divisibility and order relations. Nevertheless, only a few lemmas contain linear dependencies.

`exp-exhelp`: This example is taken from [KS96a]. It states the equivalence of call-by-value and call-by-name evaluations for simple arithmetic expressions containing function calls. Linear dependencies arise, in particular, in the termination proofs of the mutually recursive operators.

`sqrt` (H): In this case study, we prove the irrationality of $\sqrt{2}$ based on geometric ideas of Hippasus of Metapontum. The proofs contain linear and non-linear dependencies, i.e. dependencies between products.

`f91`: In this example, we prove termination of McCarthy's `f91` function [MM70]:

Example `sortalgos`						
Cfg.	Lemmas	Man. Interact.	Autom. Appl.	Del.	Fin. P.	Runtime
(A)	111	$1 + 0$	2233	40	2193	5.95
(B)	111	$1 + 0$	2200	45	2155	6.07
(C)	117	$1 + 0$	2299	63	2236	6.80
(D)	111	$1 + 0$	2253	49	2204	6.66

Example `gcd`						
Cfg.	Lemmas	Man. Interact.	Autom. Appl.	Del.	Fin. P.	Runtime
(A)	85	$8 + 2$	1114	13	1101	2.92
(B)	85	$8 + 2$	1114	13	1101	2.79
(C)	86	$8 + 2$	963	10	953	4.30
(D)	57	$8 + 2$	830	10	820	3.85

Example `exp-exhelp`						
Cfg.	Lemmas	Man. Interact.	Autom. Appl.	Del.	Fin. P.	Runtime
(A)	27	$0 + 6$	1278	116	1162	7.22
(B)	27	$0 + 6$	1275	116	1159	7.86
(C)	31	$0 + 6$	718	0	718	2.82
(D)	18	$0 + 6$	675	0	675	2.70

Example `sqrt` (H)						
Cfg.	Lemmas	Man. Interact.	Autom. Appl.	Del.	Fin. P.	Runtime
(A)	51	$11 + 1$	1062	14	1048	5.91
(B)	51	$11 + 1$	1007	14	993	5.38
(C)	49	$7 + 1$	516	27	489	2.82
(D)	18	$3 + 1$	352	23	329	1.51

Example `f91`						
Cfg.	Lemmas	Man. Interact.	Autom. Appl.	Del.	Fin. P.	Runtime
(D)	11	$0 + 3$	443	13	430	1.85

Table 4.1: Comparison of Different Configurations Within QuodLibet

$$\mathtt{f91}(m) = \begin{cases} m - 10 & \text{if } m > 100 \\ \mathtt{f91}(\mathtt{f91}(m + 11)) & \text{otherwise} \end{cases}$$

The proof is based on linear dependencies stated with two auxiliary lemmas without exploiting the fact that

$$\mathtt{f91}(m) = \begin{cases} m - 10 & \text{if } m > 100 \\ 91 & \text{otherwise} \end{cases}$$

which would simplify the proof a lot. Note that the function is defined with large constants. Therefore, we cannot prove termination without linear arithmetic.

To be able to study solely the effects of the integration, we compare the proof control with and without linear arithmetic with as few differences as possible. For all our case studies, we use the same base system in which the new inference rules have been integrated. Differences may occur in the simplification process and in the specifications that are used. Altogether, we compare four different configurations:

(A) In this configuration, we use the old waterfall which does not apply any of the new inference rules for linear arithmetic. In the specifications, we do not use the predefined sort and operators for linear arithmetic. Instead, we model natural numbers with a new sort `nat` with constructors `0` and `S` and defined operators `plus`, `minus`, `times`, and `leq`.

(B) In comparison to Configuration (A), we use the new waterfall with linear arithmetic in this and the following configurations. But in this configuration we still use the same specifications as in Configuration (A). Thus, none of the phases for linear arithmetic is applicable. But since they are checked for applicability, we expect slightly increased runtimes. The computations may differ only in that definedness atoms are preferred in the new waterfall: The handling of definedness atoms in the new phases `def1` and `def2` has been split from the handling of the other atoms in phases `reduce1` and `reduce2`.

(C) In this configuration, we replace sort `nat` and its constructors and defined operators with the predefined sort `Nat` and the predefined constructors and defined operators. Apart from this, we try to leave the specifications unchanged as far as possible. We prepend each specification only with an additional simple and uniform base specification that contains basic properties of the predefined operators such as their definedness and the associativity, commutativity and distributivity of operator `*`. Therefore, the number of lemmas may slightly increase.

(D) In the last configuration, we exploit the advantages of the integration. Therefore, we delete unused lemmas in the specification. Furthermore, we slightly modify the specifications to achieve a better automation.

In Table 4.1, we list for each example and each configuration, in column

Lemmas the number of lemmas that have to be supplied manually. Roughly speaking, the degree of automation is "inversely proportional" to this number.

Man. Interact. the number of manual interactions. More precisely, the column contains the number of manually applied inference rules and the number of manual instantiations of weight variables to choose appropriate induction orders. These values also reflect the degree of automation.

Autom. Appl. the number i of inference rules that are applied automatically during the proof (including those that are applied tentatively and deleted again).

Del. the number d of deleted applications of inference rules. The runtime essentially depends on the automatic applications and deletions.

Fin. P. the number of applied inference rules in the final proof, i.e. $i-d$. This is a measure for the complexity of the resulting proofs.

Runtime the runtime in seconds measured by a CMU COMMON LISP system on a machine with a 1 GHz Intel III processor and 4 GB RAM. Note that this measurement is not very precise. Sometimes we observed rather large deviations.

From the statistics in Table 4.1, we draw the following conclusions:

- If the specifications do not contain the interpreted symbols of linear arithmetic, then normally the runtime increases slightly for the new waterfall (cf. Configurations (A) and (B)) as long as the proofs remain the same. The case study `sqrt` (H), however, benefits from the preference of definedness atoms in the new waterfall. Thus, it generates a simpler proof with reduced runtime.
- The more suitable the case studies are for linear arithmetic the more profit we gain by using the interpreted symbols (cf. Configurations (B) and (C)). Whereas interpreted symbols result in even more complicated proofs with increased runtimes for `sortalgos`, the proofs become easier for the other case studies with significantly reduced runtimes for `exp-exhelp` and `sqrt` (H).
- In the same way, the number of auxiliary lemmas may be reduced for suitable specifications (cf. Configuration (A) and (D)).
- We may improve the degree of automation as well as the simplicity of the proofs and their runtime if we adapt the specifications to linear arithmetic (cf. Configurations (C) and (D)).
- The integration allows us to handle case studies with large constants such as `f91` that were out of scope of the old simplification process.

To summarize, for case studies where the new inference rules are hardly applicable, the efficiency of the new proof control with linear arithmetic may slightly decrease. But for the other case studies, we often get a significant speed-up and an improved degree of automation. In particular, the decreased number of auxiliary lemmas is beneficial.

4.3.2 Comparison with Other Approaches

In this and the next section, we have a closer look at six case studies (cf. Table 4.2). Five of them are taken from the literature [BM88b, KN94]. These problems were used as benchmarks for the incorporation of the *extended proof method* `EPM` into `Clam` [JB02], and *constraint contextual rewriting* `CCR(X)` into `RDL` [AR03], respectively. The last problem corresponds to `sqrt` (H) and is considered only in the next section. Table 4.2 contains for each problem the auxiliary lemmas L that are available to prove goal G.

Table 4.3 contains the runtime in seconds for the systems `Clam`, `RDL` and QUODLIBET, respectively. An entry '—' means that the test was not performed with the system, '?' means that the goal was not proved. Note that we did not perform the experiments for `Clam` and `RDL` on our own. Instead we quote the results mentioned for `Clam` in [JB02] and for `RDL` in [AR03]. The tests for `Clam` were performed on a 433 MHz PC, whereas the tests for `RDL` and QUODLIBET were made on a 1 GHz PC. Note that our measurements contain the output of a detailed proof log. From the results in Table 4.3, we can see that our integration scheme is competitive.

Prob#		Problem
1 [BM88b]	G	$\{\ L < \texttt{+}(\texttt{MAX}(A), K),\ \neg(L \le \texttt{MIN}(A)),\ \neg(\texttt{0} < K),\ A = \texttt{nil}\ \}$
	L	(i) $\{\ \texttt{MIN}(A) \le \texttt{MAX}(A),\ A = \texttt{nil}\ \}$
2 [BM88b]	G	$\{\ \texttt{+}(I, \texttt{DELTA1}(PAT, LP, C)) \le MAXINT,\ \neg(\texttt{+}(LP, LT) \le MAXINT),$ $\neg(I \le LT)\ \}$
	L	(i) $\{\ \texttt{DELTA1}(PAT, LP, C) \le LP\ \}$
3 [BM88b]	G	$\{\ \texttt{+}(W, \texttt{LEN}(\texttt{DEL}(Z, A))) < \texttt{+}(K, V),\ \texttt{MEMB}(Z, A) \ne \texttt{true},$ $\neg(\texttt{+}(W, \texttt{LEN}(A)) \le K)\ \}$
	L	(i) $\{\ \texttt{LEN}(\texttt{DEL}(X, S)) < \texttt{LEN}(S),\ \texttt{MEMB}(X, S) \ne \texttt{true}\ \}$
4 [BM88b]	G	$\{\ \texttt{+}(\texttt{+}(\texttt{MS}(c), \texttt{*}(\texttt{MS}(a), \texttt{MS}(a))), \texttt{*}(\texttt{MS}(b), \texttt{MS}(b)))$ $< \texttt{+}(\texttt{+}(\texttt{+}(\texttt{MS}(c), \texttt{*}(\texttt{MS}(b), \texttt{MS}(b))), \texttt{*}(\texttt{2}, \texttt{*}(\texttt{MS}(a), \texttt{*}(\texttt{MS}(a), \texttt{MS}(b))))),$ $\texttt{*}(\texttt{MS}(a), \texttt{*}(\texttt{MS}(a), \texttt{*}(\texttt{MS}(a), \texttt{MS}(a)))))\ \}$
	L	(i) $\{\ J \le \texttt{*}(I, J),\ \neg(\texttt{0} < I)\ \}$
		(ii) $\{\ \texttt{0} < \texttt{MS}(x)\ \}$
5 [KN94]	G	$\{\ z < \texttt{+}(\texttt{g}(x), y),\ \texttt{p}(x) \ne \texttt{true},\ \neg(z \le \texttt{f}(\texttt{max}(x, y))),\ \neg(\texttt{0} < \texttt{min}(x, y)),$ $\neg(x \le \texttt{max}(x, y)),\ \neg(\texttt{max}(x, y) \le x)\ \}$
	L	(i) $\{\ \texttt{f}(x) \le \texttt{g}(x),\ \texttt{p}(x) \ne \texttt{true}\ \}$
		(ii) $\{\ \texttt{min}(x, y) = y,\ \texttt{max}(x, y) \ne x\ \}$
6	G	$\{\ \texttt{*}(\texttt{2}, \texttt{*}(y, y)) \ne \texttt{*}(x, x),\ y = \texttt{0}\ \}$
	L	(i) $\{\ \texttt{*}(w, x) \le \texttt{*}(y, z),\ \texttt{+}(\texttt{1}, y) \le w,\ \texttt{+}(\texttt{1}, z) \le x\ \}$
		(ii) $\{\ \texttt{*}(y, y) \ne \texttt{0},\ y = \texttt{0}\ \}$

Table 4.2: Benchmark Problems

Prob#	`Clam` [433 MHz]	`RDL` [1 GHz]	QuodLibet [1 GHz]
1	0.14	—	0.03
2	0.23	0.01	0.03
3	—	0.01	0.02
4	5.73	0.03	0.23
5	?	0.06	0.10

Table 4.3: Runtimes for the Benchmark Problems (taken from [JB02] and [AR03])

P#	T.	Literals (Framed Literals are Important for Lemma Speculation)	S.L.
1	S	$+(1, L) \le +(K, \mathtt{MAX}(A))$, $+(1, \mathtt{MIN}(A)) \le L$, $K \le 0$, $\boxed{A = \mathtt{nil}}$	—
	E	$L \le \mathtt{MAX}(A)$, $\boxed{\mathtt{MIN}(A) \le \mathtt{MAX}(A)}$, $+(1, \mathtt{MIN}(A)) \le +(K, \mathtt{MAX}(A))$, ...	(i)
2	S	$+(I, \mathtt{DELTA1}(PAT, LP, C)) \le MAXINT$, $+(1, MAXINT) \le +(LP, LT)$, $+(1, LT) \le I$	—
	E	$+(LT, \mathtt{DELTA1}(PAT, LP, C)) \le MAXINT$, $+(MAXINT, \mathtt{DELTA1}(PAT, LP, C)) \le +(*(2, LP), LT)$, $+(1, MAXINT) \le +(I, LP)$, $\boxed{\mathtt{DELTA1}(PAT, LP, C) \le LP}$, $+(I, \mathtt{DELTA1}(PAT, LP, C)) \le +(LP, LT)$, ...	(i)
3	S	$+(1, +(W, \mathtt{LEN}(\mathtt{DEL}(Z, A)))) \le +(K, V)$, $\boxed{\mathtt{MEMB}(Z, A) \ne \mathtt{true}}$, $+(1, K) \le +(W, \mathtt{LEN}(A))$	—
	E	$\boxed{+(1, \mathtt{LEN}(\mathtt{DEL}(Z, A))) \le +(V, \mathtt{LEN}(A))}$, ...	(i)
4	S	$+(1, *(\mathtt{MS}(a), \mathtt{MS}(a))) \le +(*(2, *(\mathtt{MS}(a), *(\mathtt{MS}(a), \mathtt{MS}(b)))),$ $*(\mathtt{MS}(a), *(\mathtt{MS}(a), *(\mathtt{MS}(a), \mathtt{MS}(a)))))$	(i)
	S	$*(\mathtt{MS}(a), *(\mathtt{MS}(a), \mathtt{MS}(a))) \ne *(\mathtt{MS}(a), \mathtt{MS}(a))$, $\boxed{*(\mathtt{MS}(a), \mathtt{MS}(b)) \ne 0}$	(ii)
5	S	$\boxed{\mathtt{max}(x, y) \ne x}$, $+(1, z) \le +(y, \mathtt{g}(x))$, $\mathtt{p}(x) \ne \mathtt{true}$, $+(1, \mathtt{f}(x)) \le z$, $\boxed{\mathtt{min}(x, y) \le 0}$	(ii)
	S	$\mathtt{max}(x, y) \ne x$, $+(1, z) \le +(y, \mathtt{g}(x))$, $\boxed{\mathtt{p}(x) \ne \mathtt{true}}$, $+(1, \mathtt{f}(x)) \le z$, $y \le 0$	—
	E	$z \le \mathtt{g}(x)$, $\boxed{\mathtt{f}(x) \le \mathtt{g}(x)}$, $+(1, \mathtt{f}(x)) \le +(y, \mathtt{g}(x))$, ...	(i)
6	S	$\boxed{+(1, y) \le x}$, $\boxed{*(2, *(y, y)) \ne *(x, x)}$, $y = 0$	(i)
	S	$\boxed{*(y, y) \ne 0}$, $+(1, y) \le x$, $0 \ne *(x, x)$, $\boxed{y = 0}$	(ii)

Table 4.4: Speculation of Auxiliary Lemmas

4.3.3 Effects on the Speculation of Auxiliary Lemmas

In this section, we want to investigate how our close integration into QuodLibet supports the speculation of auxiliary lemmas. Thus, we consider the problems from Table 4.2 once again but without any auxiliary lemmas. We sketch the process of deriving these lemmas with QuodLibet in Table 4.4. For each problem, we list the tactics (T.) we call: the usual simplification process is represented with `S`, the special purpose tactic that performs all variable elimination steps is abbreviated with `E`. The literals of the resulting subgoal are given in the next column (without regarding (negated) definedness atoms). For the special purpose tactic `E`, we display only the new literals; the ellipses stand for the literals after the last execution of the simplification process S given in the previous line. Literals that are used to speculate a lemma are framed. The last column contains the speculated lemmas (S.L.) w.r.t. Table 4.2. The number of literals measures the complexity of the goal: The more literals are present, the more difficult it is to find the important literals, and thus an auxiliary lemma. We believe that their identification has to be done with human expertise. We assume that this task is rather easy if the auxiliary lemma of the considered problem in Table 4.2 is a subformula of the resulting subgoal clause in Table 4.4.

The first two problems do not cause any difficulties. After applying both tactics, the auxiliary lemmas are subformulas of the resulting goal clauses. This is, however, not the case if we call only the simplification process S. For Problem 1, the derivation of the subgoal can be found in Figure 4.1 (cf. Section 4.1.3). The first important literal can be identified if we look for a literal that contains at least one uninterpreted function symbol but as few extra variables as possible. In this context, by an *extra* variable of a literal we mean a variable that does not occur in a subterm of the literal with an uninterpreted function symbol as top-level symbol. The last goal node in Figure 4.1 contains the extra variable L for the first and fifth literal; K for the third and sixth literal; L and K for the fourth literal; and A for the seventh literal. But a lemma that consists only of the second literal $\mathtt{MIN}(A) \leq \mathtt{MAX}(A)$ is not inductively valid. Instead, a human expert has to add the last literal $A = \mathtt{nil}$ to get an inductively valid lemma. For Problem 3, the resulting subgoal contains an additional variable V. This can be eliminated if we use the fact that we deal with naturals only. But at the moment, this is done automatically by the simplification process S only if this seems to be advantageous. Nevertheless, if only one lemma is missing, our close integration facilitates the speculation of auxiliary lemmas quite well.

For the remaining problems, two auxiliary lemmas are missing. For Problems 4 and 6, our tactics do not provide any additional information for the first lemma. This is not very surprising since in these examples no variable elimination steps can be performed at all. Note that the first important literal for Problem 6 is introduced by a manual inductive case split. With the two important literals for the first lemma of Problem 6, it is not difficult for a human user with domain knowledge about multiplication to guess the required monotonicity property as auxiliary lemma. To speculate the first auxiliary lemma for Problem 5, an experienced user needs to know only the first important literal and the left-hand side of the second one. Then, the relationship between max and min is obvious. Note that in the original goal clause presented in Table 4.2, the first important literal is not present. This complicates the speculation of the required auxiliary lemma for the original goal. Only for Problem 4, the second auxiliary lemma is not a subformula of the considered subgoal clause. But the important literal of this problem suggests an auxiliary lemma that may be used as well.

To conclude, seven of nine lemmas can be speculated easily with our integration scheme. Four of them require an additional call to the special purpose tactic E because the simplification process S does not provide enough information. For the speculation of auxiliary lemmas, 14 of 42 literals are important. In the presented case studies there is no exponential blow-up of the number of inequalities.

4.4 Related Work

In the literature, there exist many other decision procedures for linear arithmetic besides that of Hodes [Hod71]. They differ in their principle approach as well as their underlying domain. The approaches are, for instance, based on

- eliminating variables as e.g. in Hodes' decision procedure for the rationals and in Cooper's decision procedure for the integers [Coo72].
- calculating lower and upper bounds (on rational numbers) and testing the resulting intervals for (integral) solutions as e.g. in the SUP-INF method [Ble75, Sho77].
- constructing finite automata which accept exactly the solutions of a set of constraints over linear arithmetic. Therefore, the satisfiability problem is reduced to the problem whether the language of the constructed automaton is non-empty [BC96].
- using Hodes' decision procedure to reduce linear arithmetic to boolean satisfiability. For this, satisfiable constraints are represented with boolean variables, unsatisfiable constraints with the boolean value false, and variable elimination steps with implications which relate the different constraints [Str02].
- using linear programming techniques such as the Simplex method.

In general, decision procedures for linear arithmetic over the rationals are simpler w.r.t. their time complexity than those over the integers or naturals. Furthermore, the universal fragment over the integers (PIA) or naturals (PNA)—which we are concerned with in this chapter—is simpler than the full theory. Cooper's decision procedure for the full theory over the integers, for instance, has time complexity $2^{2^{2^N}}$ in the length N of the input formula. The universal fragment can be decided with the SUP-INF method in 2^N. Even for the rationals, Hodes' decision procedure has time complexity in m^{2^n} where m is the number of constraints and n the number of variables in the input formula. But as explained in [BM88b], the efficiency of the decision procedure itself is irrelevant when using it in an extended theory. In the cited case study with `NQTHM`, an instantaneous oracle for linear inequalities would reduce the overall runtime by less than 3%. Instead, the interaction between the decision procedure and the theorem prover is more important. The advantage of Hodes' decision procedure is its simplicity. Therefore, we have not investigated other decision procedures in detail.

Our work is inspired by [BM88b], where Boyer & Moore describe many helpful heuristics to restrict the search space. But their description is sometimes hard to read as they use internal data structures special to their theorem prover `NQTHM`. Instead, we use inference rules for the incorporation.

In [KN94], Kapur & Nie extend the approach from [BM88b] at least in two ways: They do not convert equations into two inequalities but use equations directly to eliminate variables. This handling of equality information is more efficient than that in [BM88b]. Furthermore, they extend the decision procedure for PRA in such a way that it is also a decision procedure for PIA and PNA: At first, the closure under the variable elimination steps is calculated. If no unsatisfiable inequality can be found, then there exists a rational solution which is determined by the inequalities. This solution has to be checked for a solution over integers (or naturals). These improvements are realized, on the one hand, with `la-const-rewrite` and $\neq$`-var-elim`, and on the other hand, with $\leq$`-case-split`. Our automatic proof control does not realize the whole decision procedure for PNA as this may result in a huge case distinction. Instead, we use this method only if the intervals that have to be checked are small. Otherwise, we have to use other proof techniques such as induction.

Janicic, Bundy & Green [JBG99] formalize and generalize the approach from [BM88b]. Their presentation is independent from the theory and the decision procedure to be used. Instead, they assume that the decision procedure can be divided into two steps: the elimination of variables and a check on ground instances. This approach is further developed in [JB02] taking into account the combination of decision procedures. Our fragmentation of the decision procedure into inference rules is influenced by [JBG99]. As a third major step of a decision procedure, we identify the normalization of literals.

The approach proposed by Armando & Ranise [AR03] is similar to that in [JBG99]. But they combine the decision procedure more closely with rewriting. They pose additional demands on the rewriting mechanism and the decision procedure. This allows them to prove soundness and termination properties for their approach. The soundness of our approach is guaranteed by local properties of our inference rules. Termination properties may be proved by constraining the control in a similar way as in [AR03].

In [BGD03], Berezin, Ganesh & Dill propose an inference system for their integration scheme. Our inference rules are on a higher level. Therefore, they can be easier applied manually. Although their inference rules can be easier checked with an external proof checker this is also possible for ours.

[KS03] and [GK03] contain proposals for the speculation of rewrite rules. For nonlinear equations, Armando, Rusinowitch & Stratulat propose the use of Buchberger's algorithm based on Gröbner basis in [ARS02]. In [AR01] and [HKM03], approaches are described to extend the integration of linear arithmetic to nonlinear arithmetic. These extensions just improve the heuristics for choosing and speculating lemmas which may be used for the augmentation mechanism in this special domain. Therefore, the integration of these extensions into our approach should be easy by using special purpose tactics. This is subject of further research.

Chapter 5

Adaptable Inference Systems

The inference rules of QuodLibet (cf. Sections 2.2.2 and 4.2.3) are given by parameterized schemes. The applicability of an inference rule as well as the new subgoals essentially depend on *some* of the elements in the parent goal only—called the *principal elements*. Therefore, we can adapt single proof steps as well as whole proofs from one goal to another if only the "essential" elements of the former one are present in the latter one. We define the essence of a proof with the notion of *contribution*. In this chapter, we provide a first abstract definition of contribution which depends on the principal elements in the goals w.r.t. the single proof steps performed. This definition of contribution is refined w.r.t. a heuristics for selecting appropriate subgoals for non-contributing proof steps in Chapter 7 where contribution is exploited for reusing performed proofs. As a second application, *local* contribution can be used for guiding proof search (cf. Chapter 6).

The notion of contribution is not restricted to the inference system of QuodLibet but may be applied to many other reductive inference systems—i.e. those that reduce goals to new subgoals—working on proof state trees. Therefore, we abstract from the concrete inference system of QuodLibet. Instead, we introduce notions on inference systems that enable the definition of contribution and its applications such as guiding proof search and reusing proofs.

In this and the next two chapters, we provide three different views on contribution. In this chapter, we lay the foundations without considering the applications. Instead, we start in a bottom-up style: We identify commonalities among the inference rules of QuodLibet in Section 5.1. This leads to the notion of *adaptable* inference systems in Section 5.2. We conclude this chapter with a first abstract definition of contribution in Section 5.3. In the next two chapters, we make up for the applications.

5.1 Identification of Commonalities Among the Inference Rules of QuodLibet

To identify commonalities among the inference rules of QuodLibet, we first have a look at some examples of applications.

(a)

{ elemleqlist-p(n, merge(cons(m, $l1$), $l2$)) = true,
¬def merge(cons(m, $l1$), $l2$),
elemleqlist-p(n, $l2$) $\neq$ true,
elemleqlist-p(m, $l1$) $\neq$ true,
$m \leq n$,
sorted-p(cons(n, merge(cons(m, $l1$), $l2$))) = true,
sorted-p($l1$) $\neq$ true,
sorted-p($l2$) $\neq$ true }

lemma-rewrite 1 [1] (5.1) 1 [$m \leftarrow n$, $l1 \leftarrow$ cons(m, $l1$), $l2 \leftarrow l2$]

{ elemleqlist-p(n, cons(m, $l1$)) = true,
elemleqlist-p(n, merge(cons(m, $l1$), $l2$)) = true,
¬def merge(cons(m, $l1$), $l2$),
elemleqlist-p(n, $l2$) $\neq$ true,
elemleqlist-p(m, $l1$) $\neq$ true,
$m \leq n$,
sorted-p(cons(n, merge(cons(m, $l1$), $l2$))) = true,
sorted-p($l1$) $\neq$ true,
sorted-p($l2$) $\neq$ true }

{ elemleqlist-p(n, cons(m, $l1$)) $\neq$ true,
true = true,
¬def merge(cons(m, $l1$), $l2$),
elemleqlist-p(n, $l2$) $\neq$ true,
elemleqlist-p(m, $l1$) $\neq$ true,
$m \leq n$,
sorted-p(cons(n, merge(cons(m, $l1$), $l2$))) = true,
sorted-p($l1$) $\neq$ true,
sorted-p($l2$) $\neq$ true }

(b)

{ elemleqlist-p(n, merge(cons(m, $l1$), $l2$)) = true,
elemleqlist-p(n, $l2$) $\neq$ true }

lemma-rewrite 1 [1] (5.1) 1 [$m \leftarrow n$, $l1 \leftarrow$ cons(m, $l1$), $l2 \leftarrow l2$]

{ elemleqlist-p(n, cons(m, $l1$)) = true,
elemleqlist-p(n, merge(cons(m, $l1$), $l2$)) = true,
elemleqlist-p(n, $l2$) $\neq$ true }

{ elemleqlist-p(n, cons(m, $l1$)) $\neq$ true,
true = true,
elemleqlist-p(n, $l2$) $\neq$ true }

(c)

{ elemleqlist-p(n, cons(m, $l1$)) $\neq$ true,
elemleqlist-p(m, $l1$) $\neq$ true,
elemleqlist-p(n, merge(cons(m, $l1$), $l2$)) = true,
¬def merge(cons(m, $l1$), $l2$),
elemleqlist-p(n, $l2$) $\neq$ true }

lemma-rewrite 3 [1] (5.1) 1 [$m \leftarrow n$, $l1 \leftarrow$ cons(m, $l1$), $l2 \leftarrow l2$]

{ elemleqlist-p(n, cons(m, $l1$)) $\neq$ true,
elemleqlist-p(m, $l1$) $\neq$ true,
true = true,
¬def merge(cons(m, $l1$), $l2$),
elemleqlist-p(n, $l2$) $\neq$ true }

Figure 5.1: Three Applications of Lemma (5.1) for Rewriting

Example 5.1 We assume that we have entered a specification containing a lemma

$$\{\ \texttt{elemleqlist-p}(m, \texttt{merge}(l1, l2)) = \texttt{true},\ \texttt{elemleqlist-p}(m, l1) \neq \texttt{true},\ \texttt{elemleqlist-p}(m, l2) \neq \texttt{true}\ \} \tag{5.1}$$

Figure 5.1 illustrates its application to three different goals. The first application in Figure 5.1(a) is taken from a case study about mergesort. The applicability of Lemma (5.1) for rewriting depends only on the first literal in the goal. This focus literal (cf. Section 2.2.2.10) is rewritten with the lemma. The number of new subgoals is additionally influenced by the third literal in the goal which cuts off one of the new condition subgoals because it directly fulfills the last literal of the lemma. Thus, the first and the third literal are essential for the application. We call these goal literals *principal* for the application according to [Gen35]. The other literals in the goal are left unchanged by the application. They are passively inherited to the new subgoals. We call these goal literals *context* literals.

If we extract the essence from the goal w.r.t. the application of Lemma (5.1), i.e., if we consider a goal that contains only the two principal literals, the lemma is still applicable. Its application results essentially in the same new subgoals (cf. Figure 5.1(b)). The same literals are generated in the new subgoals by the principal literals. The empty context does not generate any further literals.

Lemma (5.1) is also applicable to the goal in Figure 5.1(c) since it contains the principal literals for the application. In general, additional literals in the goal may result in fewer new subgoals because they may be used as cut-off literals. In this case, the first literal is an additional cut-off literal which cuts off the first new subgoal. Note that the literal that is rewritten is in third place. Therefore, we have to *adapt* the application of the inference rule which refers to the position of this literal in the goal clause.

The new subgoal in Figure 5.1(c) is similar to the second new subgoal in Figures 5.1(a) and 5.1(b):

- The literals generated by the principal literals in the second new subgoal of Figures 5.1(a) and 5.1(b) are also present in the new subgoal of Figure 5.1(c).
- The common context literals—such as $\neg\texttt{def merge}(\texttt{cons}(m, l1), l2)$ in Figures 5.1(a) and Figure 5.1(c)—generate the same new literals in the new subgoals.

□

Example 5.2 Figure 5.2 illustrates the application of inference rule $\neq$`-unif`. The example is also taken from our case study about mergesort. The applicability of the inference rule depends only on the first literal. Therefore, it is the only principal literal for the application. In this case, the context literals are modified in a uniform way by applying a substitution that is solely determined by the principal literal. Once again, if we apply the inference rule to another goal with the same principal literal, then common context literals in both goals will generate the same new literals in the new subgoals. □

These two examples are typical for the inference rules of QuodLibet. The literals in the goal clause may be partitioned into *principal* and *context* literals w.r.t. the applied inference

$\{\ l \neq \texttt{nil},$
$\quad \texttt{split2}(l) < l,$
$\quad \texttt{cons}(n, \texttt{split2}(l)) < \texttt{cons}(n, l),$
$\quad \texttt{cons}(n, \texttt{split2}(l)) < \texttt{cons}(m, \texttt{cons}(n, l))\ \}$

$\neq$`-unif 1`

$\{\ \texttt{split2}(\texttt{nil}) < \texttt{nil},$
$\quad \texttt{cons}(n, \texttt{split2}(\texttt{nil})) < \texttt{cons}(n, \texttt{nil}),$
$\quad \texttt{cons}(n, \texttt{split2}(\texttt{nil})) < \texttt{cons}(m, \texttt{cons}(n, \texttt{nil}))\ \}$

Figure 5.2: Application of Inference Rule $\neq$`-unif`

rule. On the one hand, those literals that determine the applicability of an inference rule must be principal. In QuodLibet, the positions of these literals have to be provided as parameters of the inference rule. On the other hand, the fewer literals are principal the more flexibility for guiding proof search we get and the more reuse of proofs becomes possible. Both for heuristic guidance and reuse of proofs, it has turned out to be appropriate, however, that the principal literals should also include those literals that just cut off certain subgoals. We call the latter kind of literals *cut-off* literals.

For the following rules in QuodLibet cut-off literals are taken into account automatically: $=$`-decomp`, `def-decomp`, $<$`-decomp`, $\neq$`-removal`, $\neg$`def-removal`, `lemma-rewrite`, `lemma-subs`, `axiom-rewrite`, `axiom-subs`, `appl-lit-removal`, `ind-rewrite`, `ind-subs`, and all the new derived inference rules in Section 4.2.3. Basically, the occurrence of additional cut-off literals just results in the deletion of some subgoals in the proof state tree. Missing cut-off literals, however, would result in additional subgoals that may cause a failure of the proof, both when searching for or reusing it. Thus, cut-off literals should be considered principal literals. A special treatment of cut-off literals in comparison to other principal literals is rarely needed. In fact, for the applications presented in this thesis, we need such a special treatment only for modeling Contextual and Case Rewriting with a forbidden marking as presented in Section 6.2.2. Therefore, we consider the distinction between cut-off literals and other principal literals only as a last add-on in Definition 5.6.

We consider inference systems that work on goals. In the following, we state more precisely how the new subgoals are generated w.r.t. the principal part and the context. This is done with a couple of functions illustrated in Figure 5.3. The notions are partially inspired by Gentzen's notions on sequent calculi [Gen35]. The meaning of the functions is described in an informal way in this section. The requirements on these functions are formally defined in Section 5.2 with the notion of *adaptable* inference systems.

An (instance of an) *inference rule*[1] I is applied to a *parent* goal $PG(I)$. The application results in $nbSGs(I)$ new subgoals $SGs(I)$. To enhance readability, in the illustration on the left-hand side of Figure 5.3, we assume that the application of inference rule I to parent goal G results in exactly two new subgoals. For our approach it is essential that the applicability of I to G does not depend on the whole goal but only on a part of it—called the *principal* part ($princ(I) \subseteq G$). The remaining part of the goal is called *context*

[1]In the following, we consider only instances of inference rules. For brevity, we call these instances just inference rules.

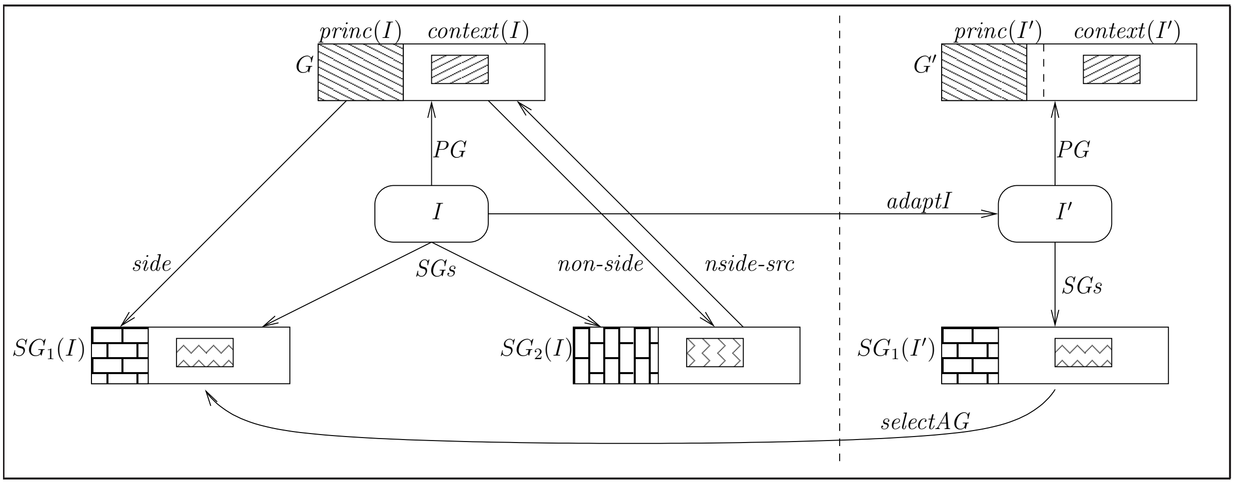

Figure 5.3: Illustration of the Notions for Adaptable Inference Systems

($context(I) \subseteq G$). Furthermore, the new subgoals must not be generated in an arbitrary way but their derivation from G is fixed. We require that we get similar new subgoals if we adapt inference rule I to another goal that contains the principal part $princ(I)$, resulting in the inference rule I' on the right-hand side of Figure 5.3. The principal part determines the essential appearance of the new subgoals. In the subgoals, some elements of the principal part may be removed or changed, new elements may be added. We model this by replacing the whole principal part with a *new* part in each new subgoal. This new part is generated by function *side*. Furthermore, the principal part determines how the context is changed in the new subgoals. Often the context is transferred to the new subgoals without modifications but it may also be changed in a uniform way e.g. by applying a substitution to each element. We model this modification with a monotonic function *non-side*. This modification has to be done in such a way that we can determine the *source* within the context that generates certain elements in the *non-side* part of a new subgoal, i.e. that part in the new subgoal that is generated with function *non-side*. This is done with function *nside-src*.

To state the adaptation of an inference rule more precisely, we use functions *adaptI* and $selectAG$. An inference rule I with parent goal G can be adapted to goal G' if G' contains the principal part of G (for I). Function *adaptI* returns an adapted inference rule I' that can be applied to G'. The principal part $princ(I') \subseteq G'$ may contain additional elements in comparison to $princ(I) \subseteq G$, illustrated with a dashed line in goal G' in Figure 5.3. We require only that for each new subgoal of I', function *selectAG* selects a subgoal of I that generates the same elements for the common parts of G and G'. In Figure 5.3, this is illustrated by hatching the common parts of the goals. In QuodLibet, the adaptation of an inference rule is achieved by possibly changing those parameters of the inference rule that contain the positions of the principal literals in the parent goal.

5.2 Adaptable Inference Systems: Formal Definitions

In this section, we present abstract requirements on inference systems that allow the definition of contribution and its applications. These requirements are defined in four steps using the functions described in Section 5.1 (cf. Figure 5.3).

Definition 5.3 (Inference Systems) An inference system $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ consists of a set $\mathcal{I}$ and two functions PG and SGs. The elements $I \in \mathcal{I}$ are called *inference rules*, the goal $PG(I)$ is called *parent goal* of I and the (possibly empty list of) finitely many goals $SGs(I)$ are called *(new) subgoals* of I. Let $nbSGs(I)$ be the number of new subgoals of I, and $SG_i(I)$ be the ith subgoal in $SGs(I)$ for $i \in \{1, \ldots, nbSGs(I)\}$. □

An inference rule I with parent goal G and new subgoals $SG_1, \ldots, SG_n$ is often represented as

$$I = \frac{G}{SG_1 \quad \ldots \quad SG_n}.$$

For Definitions 5.4 to 5.6, we assume that the following operations are defined on goals in a suitable way: A subset relation $\subseteq$, a union $\cup$, an intersection $\cap$, a sum $+$, a difference $-$ operation, and a partition relation $\uplus$ (cf. the corresponding operations on sets and multisets in Section 2.1). A goal G may, for instance, consist of n components $\langle g_1; \ldots; g_n \rangle$. Many inference systems just have one component—the clause Γ to be proved. QuodLibet uses a weight w as second component to explicitly represent the induction order. Components may consist of sets, multisets or single elements. We may define the operations on goals by defining them on the n components of the goals using the usual definitions for sets and multisets. If the component consists of a single element, we may interpret it as singleton set containing this element. This is done for the weight component in QuodLibet.

The operations on goals are defined in a suitable way if the following properties hold true for goals $G_1, \ldots, G_n$ and G:

$$\textstyle\bigcup_{i=1}^n G_i \subseteq G \quad \text{if for each } i \in \{1, \ldots, n\} : G_i \subseteq G \tag{5.2}$$
$$G_j \subseteq \textstyle\bigcup_{i=1}^n G_i \text{ for each } j \in \{1, \ldots, n\} \tag{5.3}$$
$$G \subseteq \textstyle\bigcap_{i=1}^n G_i \quad \text{if for each } i \in \{1, \ldots, n\} : G \subseteq G_i \tag{5.4}$$
$$\textstyle\bigcap_{i=1}^n G_i \subseteq G_j \text{ for each } j \in \{1, \ldots, n\} \tag{5.5}$$
$$G_1 + G_2 = G_2 + G_1 \tag{5.6}$$
$$G_1 + G_3 \subseteq G_2 + G_3 \quad \text{if } G_1 \subseteq G_2 \tag{5.7}$$
$$G_1 - G_3 \subseteq G_2 - G_3 \quad \text{if } G_1 \subseteq G_2 \tag{5.8}$$
$$G_1 \subseteq G_2 + G_3 \quad \text{iff} \quad G_1 - G_2 \subseteq G_3 \tag{5.9}$$
$$G = G_1 + G_2 \quad \text{if } G = G_1 \uplus G_2 \tag{5.10}$$
$$(G_1 + G_2) - G_1 = G_2 \quad \text{if there exist } G_3, G_4 \text{ with } G_3 = G_1 \uplus G_4 \text{ and } G_2 \subseteq G_4 \tag{5.11}$$

Note that these properties hold true for the usual definitions of the operations on sets and multisets as presented in Section 2.1. Property (5.11), for instance, holds true for multisets even unconditionally. For sets, condition $G_3 = G_1 \uplus G_4$ implies that G_1 and G_4 do not have a common element. Due to condition $G_2 \subseteq G_4$, this also holds true for G_1 and G_2. Therefore, the difference $(G_1 + G_2) - G_1$ eliminates exactly the elements in G_1 but none of the elements in G_2. Thus, $(G_1 + G_2) - G_1 = G_2$ holds true for sets.

Usually, the new subgoals $SG_1, \ldots, SG_n$ of the inference rule I are derived from parent goal G by possibly adding a new part, changing some old part, or removing some old part. According to Section 5.1 we partition G into two parts: the principal part and the context (cf. Definition 5.4(a)). The generation of the new part that replaces the principal part is modeled with function *side*. The transformation of the context is modeled with a monotonic function *non-side* (cf. Definition 5.4(b1) and (b2)). For the definition of *essential contribution* (cf. Section 7.2), we also require a function *nside-src*. This function computes those elements—the *source*—within the context that are responsible for the generation of a given set of elements in the *non-side* part of a new subgoal (cf. Definition 5.4(b3)).

Definition 5.4 (Inference Systems Defined with Principal Part and Context)
An inference system $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ is *defined with principal part and context* if there exist functions *princ*, *context*, *side*, *non-side* and *nside-src* such that the following properties are fulfilled for each inference rule $I \in \mathcal{I}$:

(a) $PG(I) = princ(I) \uplus context(I)$;

(b) and for each $i \in \{1, \ldots, nbSGs(I)\}$,

  (1) function *non-side* is monotonic in the last argument, i.e., for each pair of goals G, G' with $G \subseteq G' \subseteq context(I)$:
  $non\text{-}side(I, i, G) \subseteq non\text{-}side(I, i, G')$;

  (2) $SG_i(I)$ is generated w.r.t. *side*, *non-side* as follows:
  $SG_i(I) = side(I, i) + non\text{-}side(I, i, context(I))$;

  (3) function *nside-src* is defined such that for each $SG' \subseteq non\text{-}side(I, i, context(I))$:

  - $nside\text{-}src(I, i, SG') \subseteq context(I)$ and
  - $non\text{-}side(I, i, nside\text{-}src(I, i, SG')) = SG'$.

□

Note that every inference system can be defined with principal part and context: We may define $princ(I) = PG(I)$, $context(I) = \varnothing$, $side(I, i) = SG_i$, $non\text{-}side(I, i, \varnothing) = \varnothing$, and $nside\text{-}src(I, i, \varnothing) = \varnothing$ for each $i \in \{1, \ldots, nbSGs(I)\}$. But, as already explained in Section 5.1, to enhance the reusability of proofs we should define the principal part with as few elements as possible (cf. Section 7.3).

For adaptable inference systems, we also require that, if there is an inference rule I and a goal G' that contains the principal part of I, then there exists an inference rule I' with parent goal G'. Furthermore, each of the new subgoals of I' "corresponds" to a new subgoal of I. Thus, the same elements in the new subgoals are generated for the common part of G and G'. We use function *adaptI* to perform the adaptation of an inference rule to another goal (cf. Definition 5.5(a)). Function *selectAG* relates the new subgoals of the two applications, i.e. for each subgoal generated by inference rule I', one of the subgoals generated by I is selected that is a candidate for further adaptations since the same elements are generated for the common part of G and G' (cf. Definition 5.5(b)). Therefore, AG stands for "adaptable goal".

Definition 5.5 (Adaptable Inference Systems) Let $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ be an inference system defined with principal part and context as in Definition 5.4. $\mathcal{IS}$ is *adaptable* if there exist functions *adaptI* and *selectAG* such that the following properties are fulfilled for each inference rule $I \in \mathcal{I}$ and for each goal G' with $princ(I) \subseteq G'$:

(a) $adaptI(I, G') = I'$ such that $I' \in \mathcal{I}$ and $PG(I') = G'$;

(b) let $G'' := (PG(I) \cap G') - princ(I)$ be the common non-principal part of $PG(I)$ and G'; then, for each $SG' \in SGs(I')$:

- $selectAG(I, I', SG') \in \{1, \ldots, nbSGs(I)\}$;
- $side(I, i) + non\text{-}side(I, i, G'') \subseteq SG'$ where $i = selectAG(I, I', SG')$.

□

For the definition of (essential) contribution (cf. Sections 5.3 and 7.2), the inference system only has to be defined with principal part and context. For reusing proofs (cf. Section 7.3), however, the inference system has to be adaptable.

As a last add-on to adaptable inference systems, we consider cut-off elements which are needed only for modeling Contextual Rewriting and Case Rewriting with a forbidden marking in Section 6.2.2. The underlying idea of cut-off elements is that they have no bearing on the applicability of an inference rule but they have a bearing on the number of generated subgoals. Thus, if we have an inference rule I with parent goal G, and if goal G' results from G by eliminating all the cut-off elements in G, then there exists an inference rule I' such that G' is the parent goal of I' and I results from adapting I' to G.

Definition 5.6 (Adaptable Inference Systems with Cut-Off Part)
Let $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ be an adaptable inference system as in Definition 5.5. $\mathcal{IS}$ is an adaptable inference system with *cut-off* part if there exist functions *cut-off* and *adaptI-src* such that the following properties are fulfilled for each inference rule $I \in \mathcal{I}$:

(a) $cut\text{-}off(I) \subseteq princ(I)$;

(b) $adaptI\text{-}src(I) \in \mathcal{I}$;

(c) $PG(adaptI\text{-}src(I)) = PG(I) - cut\text{-}off(I)$;

(d) $princ(adaptI\text{-}src(I)) \subseteq princ(I)$;

(e) $adaptI(adaptI\text{-}src(I), PG(I)) = I$.

□

Lemma 5.7 The inference system of QuodLibet is an adaptable inference system with cut-off part. □

We skip the simple but lengthy proof. Instead, we present only the idea for defining the corresponding functions and illustrate the proof with an example:

- The identification of the principal and cut-off literals has already been sketched in Section 5.1. The weight is principal iff we apply a lemma as induction hypothesis.

- For inference rules $\neq$`-unif` and `subst-add`, function *non-side* applies an appropriate substitution σ_i to the context, i.e. $\mathit{non\text{-}side}(I, i, G) = G\sigma_i$ for each $G \subseteq \mathit{context}(I)$. In this case, if substitution application is not injective, function *nside-src* may pick an arbitrary inverse of *non-side* such that Definition 5.4(b3) is fulfilled.

- For all other inference rules, $\mathit{non\text{-}side}(I, i, G) = G$ for each $G \subseteq \mathit{context}(I)$, and $\mathit{nside\text{-}src}(I, i, SG') = SG'$ for each $SG' \subseteq \mathit{non\text{-}side}(I, i, \mathit{context}(I))$.

- Function *side* is defined w.r.t. function *non-side* as $\mathit{side}(I, i) = SG_i(I) - \mathit{non\text{-}side}(I, i, \mathit{context}(I))$ for each $i \in \{1, \ldots, \mathit{nbSGs}(I)\}$.

- Functions *adaptI* and *adaptI-src* adapt the parameters of the inference rule that contain the positions of the principal literals in the goal.

- Function *selectAG* chooses an appropriate new subgoal taking into account new cut-off literals.

We illustrate the proof of Lemma 5.7 with the instance of inference rule `lemma-rewrite` (cf. Section 2.2.2.10) as given in Example 5.1 (cf. Figure 5.1(a) and (c) on Page 108).

Example 5.8 Let G be the root goal node in Figure 5.1(a), I be the inference rule applied to it, $SG_1(I)$ and $SG_2(I)$ be the resulting condition subgoal on the left-hand side and the rewrite subgoal on the right-hand side, respectively.[2] As already mentioned in Example 5.1, the first and the third literal in G, i.e.

$$\texttt{elemleqlist-p}(n, \texttt{merge}(\texttt{cons}(m, l1), l2)) = \texttt{true} \quad \text{and}$$
$$\texttt{elemleqlist-p}(n, l2) \neq \texttt{true}$$

are principal for the application of I. More precisely, the third literal is a cut-off literal as it cuts off one condition subgoal. The remaining literals—i.e. the second, fourth, fifth, sixth, seventh and eighth literal—form the context of the lemma application.

The context literals are passively inherited to the new subgoals, i.e. without any modifications. Thus, $\mathit{non\text{-}side}(I, i, G_1) = G_1$ for each $G_1 \subseteq \mathit{context}(I)$; and $\mathit{nside\text{-}src}(I, i, SG) = SG$ for each $SG \subseteq \mathit{non\text{-}side}(I, i, \mathit{context}(I)) = \mathit{context}(I)$.

The other literals in the new subgoals are generated by function *side*, i.e. $\mathit{side}(I, i) = SG_i(I) - \mathit{non\text{-}side}(I, i, \mathit{context}(I))$. In the rewrite subgoal $SG_2(I)$, e.g., the first, second and fourth literal, i.e.

$$\texttt{elemleqlist-p}(n, \texttt{cons}(m, l1)) \neq \texttt{true},$$
$$\texttt{true} = \texttt{true} \quad \text{and}$$
$$\texttt{elemleqlist-p}(n, l2) \neq \texttt{true}$$

are generated by function *side*.

[2] Since the non-inductive application of a lemma with inference rule `lemma-rewrite` does not depend on and does not modify the weight we ignore it in this example and concentrate on the goal clause. Technically, the weight belongs to the context.

Let us check the conditions of Definition 5.4 for this lemma application:

(a) Obviously, the classification of the goal clause into principal part and context described above partitions the goal clause into two disjunctive lists of literals.

(b) For each $i \in \{1, 2\}$, we get

  (1) for each pair of goals G_1, G_2 with $G_1 \subseteq G_2 \subseteq \mathit{context}(I)$:
  $\mathit{non\text{-}side}(I, i, G_1) = G_1 \subseteq G_2 = \mathit{non\text{-}side}(I, i, G_2)$.

  (2) $SG_i(I) = \mathit{side}(I, i) + \mathit{non\text{-}side}(I, i, \mathit{context}(I))$ holds true because of the definition of function *side* above.

  (3) for each $SG \subseteq \mathit{non\text{-}side}(I, i, \mathit{context}(I)) = \mathit{context}(I)$:
   - $\mathit{nside\text{-}src}(I, i, SG) = SG \subseteq \mathit{context}(I)$ and
   - $\mathit{non\text{-}side}(I, i, \mathit{nside\text{-}src}(I, i, SG)) = \mathit{non\text{-}side}(I, i, SG) = SG$.

To illustrate the compliance with Definition 5.5 with a concrete example, we consider the root goal node in Figure 5.1(c) which we abbreviate with G'. Since G' contains the principal literals $\mathit{princ}(I)$ for the application of inference rule I to G as third and fifth literal, the inference rule may be adapted to G'. For this, we check the conditions of Definition 5.5:

(a) The inference rule I is adapted to G' by modifying the parameter which refers to the position of the focus literal. This parameter is set to 3 instead of 1. Then, the adapted inference rule I' is applicable to G'.

(b) Let G'' be the common part of G and G' without those literals that are principal for the application of I. G'' consists of the second and fourth literal of G which are the fourth and the second literal of G':

  $\neg\mathtt{def}\ \mathtt{merge}(\mathtt{cons}(m, \mathit{l1}), \mathit{l2})$ and
  $\mathtt{elemleqlist\text{-}p}(m, \mathit{l1}) \neq \mathtt{true}$

  As G' contains another cut-off literal for the application of I'—namely, the first one—only one new subgoal is generated—the rewrite subgoal $SG_1(I')$.

  - Function *selectAG* associates the two rewrite subgoals, i.e. $\mathit{selectAG}(I, I', SG_1(I')) = 2 \in \{1, 2\}$.
  - $\mathit{side}(I, 2)$ contains the first, second and fourth literal; $\mathit{non\text{-}side}(I, 2, G'')$ the third and fifth literal of $SG_2(I)$. These literals, i.e.

    $\mathtt{elemleqlist\text{-}p}(n, \mathtt{cons}(m, \mathit{l1})) \neq \mathtt{true}$,
    $\mathtt{true} = \mathtt{true}$,
    $\mathtt{elemleqlist\text{-}p}(n, \mathit{l2}) \neq \mathtt{true}$,
    $\neg\mathtt{def}\ \mathtt{merge}(\mathtt{cons}(m, \mathit{l1}), \mathit{l2})$ and
    $\mathtt{elemleqlist\text{-}p}(m, \mathit{l1}) \neq \mathtt{true}$

    are present in $SG_1(I')$ as first, third, fifth, fourth, and second literal. Therefore, $\mathit{side}(I, 2) + \mathit{non\text{-}side}(I, 2, G'') \subseteq SG_1(I')$ holds true.

At last, we check the compliance with Definition 5.6 for the application of inference rule I to goal G:

(a) As already mentioned above, *cut-off*(I) consists of the third literal of G which is also principal. Let G'' be derived from G by eliminating this cut-off literal.

(b) An inference rule I'' with the same parameters as I may be applied to G''.

(c) The application of I'' to G'' is assigned to I by function *adaptI-src*.

(d) Now the focus literal of the lemma application, i.e. the first literal in G'', is the only principal literal for I'' and it is also principal for I.

(e) The application of I'' to G'' generates one additional condition subgoal in comparison to the application of I to G. Since the principal literals for application I'' are present in G the inference rule may be adapted to G again. This adaptation leaves the parameters unchanged resulting in inference rule I.

□

This argumentation can be transferred to all inference rules of QuodLibet resulting in a proof for Lemma 5.7.

5.2.1 Representation of Proof Attempts with Proof State Trees

We assume that proof attempts are represented with proof state trees as described for QuodLibet in Section 2.2.2.2. A proof state tree consists of goal and inference nodes which refer to goals and inference rules. The nodes additionally contain information about their position in the tree, i.e. their parents and children. We will abuse these notions and identify goals with goal nodes and inference rules with inference nodes.

To simplify the presentation, we restrict ourselves to single proof attempts where for each goal at most one proof step is performed. The extension for arbitrary proof state trees is straightforward. Let $RG(P)$ be the root goal of proof state tree P. Let $SI(G)$ be the successor inference rule applied to G in proof state tree P (as long as there is one) and $PI(G)$ the parent inference rule of G (unless G is the root goal of P).

According to Section 2.2.2.2, a proof attempt is *closed* if there are no open goal nodes in the proof attempt, i.e. if all its leaves are inference nodes. In our abstract setting of adaptable inference systems, we call a closed proof attempt for a goal G just a *proof*[3] for G. In this case, goal G is *proved*.

Most of the properties of our approaches presented in the next two chapters are proved by induction on proof state trees. More precisely, they are proved by induction w.r.t. the following order $\prec_P$: $N_1 \prec_P N_2$ if N_1 is an offspring of N_2 in a proof state tree P. Note that $\prec_P$ is a wellfounded order on goal and inference nodes.

[3]Note that this notion of "proof" slightly differs from the notion in QuodLibet as defined in Section 2.2.2.2 where additionally all non-inductively applied lemmas have to be inductively valid. This relaxed notion suffices for the applications of contribution presented in the following chapters.

5.3 Contribution

Analyzing a proof performed with an adaptable inference system, a *proof step* I (i.e. the application of inference rule I) may *contribute* to a proof for a goal in two ways: Firstly, no new subgoals are created at all; thus, the goal is proved. Secondly, each subgoal contains new information in the form of *new* (i.e. added or changed) elements that are needed for the proof (i.e. become principal in a subsequent contributing proof step). Otherwise, the proof step is *non-contributing* and can be eliminated: If one subgoal can be proved without using one of the new elements, this proof can also be adapted to the original goal.

For the definition of contribution, we first state the phrase that an "element becomes principal in a subsequent proof step" more precisely. It depends on the performed proof attempt. Furthermore, we have to cope with the fact that the elements of the context may be changed with function *non-side* when transferred to the new subgoals whereas the principal elements are replaced with a new part generated by function *side*. Therefore, we define an inheritance relation on the elements in the goals of a proof attempt w.r.t. functions *side* and *non-side*.

Definition 5.9 (Inherited Elements in Goals of Proof State Trees)
Let $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ be an inference system defined with principal part and context as in Definition 5.4.

(a) Let I be an inference rule and $G' \subseteq PG(I)$ be a subset of the parent goal of I. For $i \in \{1, \ldots, nbSGs(I)\}$, we define the inheritance of G' from parent goal $PG(I)$ to the ith subgoal $SG_i(I)$ as follows:

$$inh(I, i, G') = \begin{cases} non\text{-}side(I, i, G') & \text{if } G' \cap princ(I) = \varnothing \\ side(I, i) + non\text{-}side(I, i, G' - princ(I)) & \text{otherwise} \end{cases}$$

(b) Let P be a proof state tree. We define an inheritance relation $\xrightarrow{*}_{inh}$ on pairs of goals[4] as the reflexive-transitive closure that contains $(PG(I), G') \longrightarrow_{inh} (SG_i(I), inh(I, i, G'))$ for each proof step I in P, each $i \in \{1, \ldots, nbSGs(I)\}$, and each subset $G' \subseteq PG(I)$.

An element λ_1 in goal G_1 *inherits* to an element λ_2 in goal G_2 if $(G_1, \{\lambda_1\}) \xrightarrow{*}_{inh} (G_2, G'_2)$ with λ_2 in G'_2.

□

Definition 5.10 (Contributing Proof Steps / Elements) Let $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ be an inference system defined with principal part and context as in Definition 5.4. A proof step I in proof state tree P for goal G *contributes* to P if every direct subgoal SG in $SGs(I)$ contains a new element that contributes to the proof state tree for SG in P. An element of a goal G *contributes* to a proof state tree P for G if it inherits to an element in the principal part of a contributing proof step in P. □

[4] More precisely, the first component is a goal node which contains a position within the proof state tree whereas the second one is goal without a position.

We aim at

- guiding proof search to avoid non-contributing proof steps (cf. Chapter 6); and
- removing non-contributing proof steps to enhance the reusability of proofs (cf. Chapter 7). Furthermore, with the deletion of non-contributing proof steps some of the proof obligations may become redundant.

We illustrate our notions of inheritance and contribution and motivate our applications of guiding proof search and reusing proofs with an extended version of Example 5.2.

Example 5.11 We consider a proof for the root goal node of Figure 5.2 on Page 110 which is displayed in Figure 5.4. After applying inference rule $\neq$-`unif` as in Example 5.2 the proof is completed with applications of Axiom (5.12) and inference rule $<$-`decomp`.

$$\{\ \texttt{split2}(\texttt{nil}) = \texttt{nil}\ \} \tag{5.12}$$

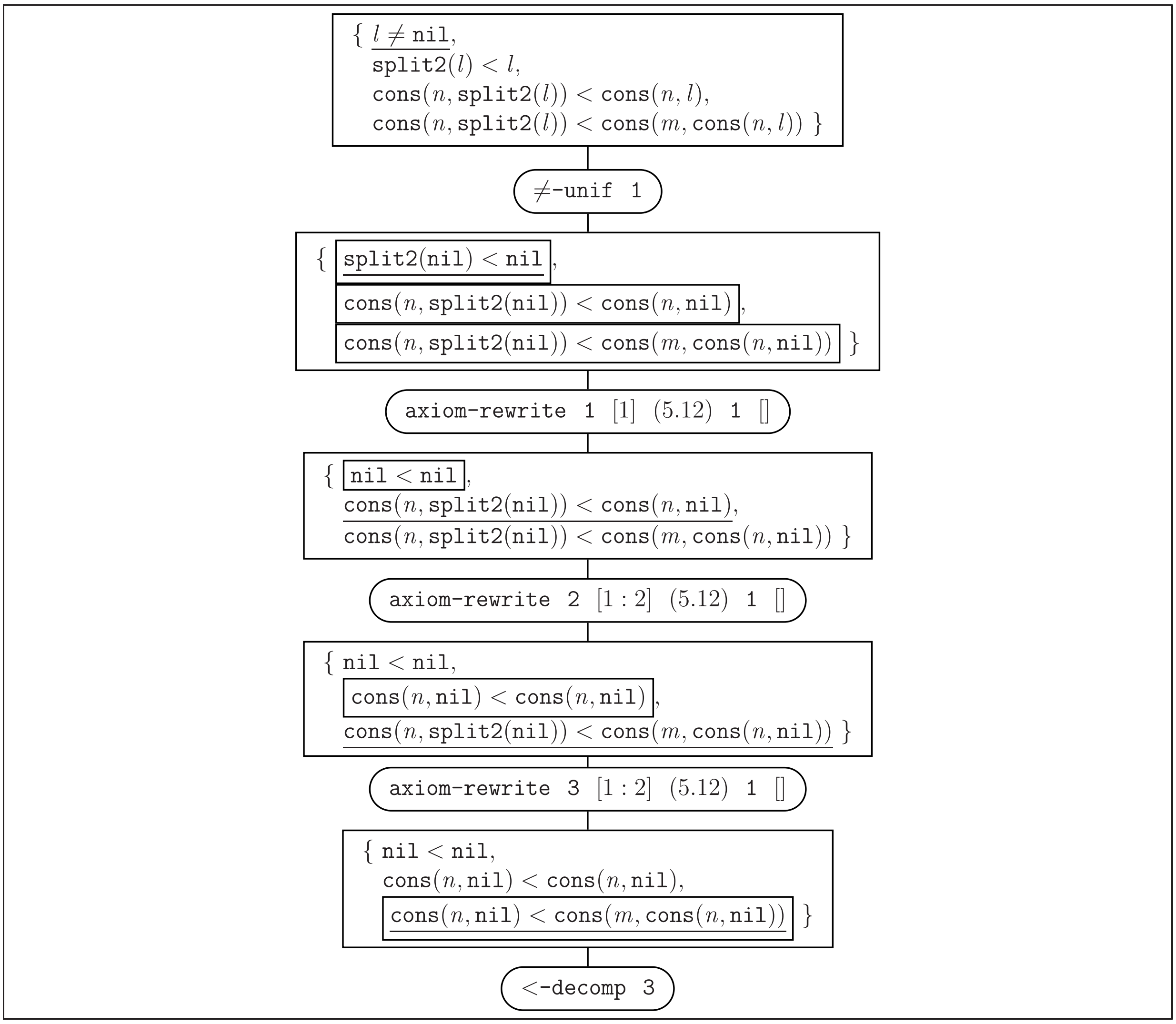

Figure 5.4: Illustration of Inheritance and Contribution

Let us calculate the contributing proof steps and elements for this proof according to Definition 5.10. For this, we have to identify the principal literals for the applied inference rules, the new literals in the resulting subgoals, and we have to consider the inheritance relation as given in Definition 5.9.

We enumerate the goal and inference nodes in Figure 5.4 top-down: Let G_1 be the root goal node, I_i be the inference rule applied to G_i for $i \in \{1, \ldots, 5\}$, and G_{i+1} be the resulting subgoal for $i \in \{1, \ldots, 4\}$.

The first parameter of each inference rule in Figure 5.4 refers to the principal literal for the application in the parent goal. Principal literals are underlined whereas new literals in the resulting subgoals are framed in Figure 5.4. Note that there are no cut-off literals in this example.

In this example, the inheritance relation is rather simple: The first literal in G_1 is eliminated, i.e. it does not inherit to any literal in G_2. Each other literal inherits to exactly one literal in each subgoal. More precisely, the second (resp. third and fourth) literal of G_1 inherits to the first (resp. second and third) literal in each other goal G_i, $i \in \{1, \ldots, 5\}$. Thus, the inheritance relation for single literals essentially consists of

$$
\begin{aligned}
&(G_1, \{l \neq \mathtt{nil}\}) \\
&\qquad \longrightarrow_{inh} (G_2, \varnothing) \\
&(G_1, \{\mathtt{split2}(l) < l\}) \\
&\qquad \longrightarrow_{inh} (G_2, \{\mathtt{split2}(\mathtt{nil}) < \mathtt{nil}\}) \\
&\qquad \longrightarrow_{inh} (G_3, \{\mathtt{nil} < \mathtt{nil}\}) \\
&\qquad \longrightarrow_{inh} (G_4, \{\mathtt{nil} < \mathtt{nil}\}) \\
&\qquad \longrightarrow_{inh} (G_5, \{\mathtt{nil} < \mathtt{nil}\}) \\
&(G_1, \{\mathtt{cons}(n, \mathtt{split2}(l)) < \mathtt{cons}(n, l)\}) \\
&\qquad \longrightarrow_{inh} (G_2, \{\mathtt{cons}(n, \mathtt{split2}(\mathtt{nil})) < \mathtt{cons}(n, \mathtt{nil})\}) \\
&\qquad \longrightarrow_{inh} (G_3, \{\mathtt{cons}(n, \mathtt{split2}(\mathtt{nil})) < \mathtt{cons}(n, \mathtt{nil})\}) \\
&\qquad \longrightarrow_{inh} (G_4, \{\mathtt{cons}(n, \mathtt{nil}) < \mathtt{cons}(n, \mathtt{nil})\}) \\
&\qquad \longrightarrow_{inh} (G_5, \{\mathtt{cons}(n, \mathtt{nil}) < \mathtt{cons}(n, \mathtt{nil})\}) \\
&(G_1, \{\mathtt{cons}(n, \mathtt{split2}(l)) < \mathtt{cons}(m, \mathtt{cons}(n, l))\}) \\
&\qquad \longrightarrow_{inh} (G_2, \{\mathtt{cons}(n, \mathtt{split2}(\mathtt{nil})) < \mathtt{cons}(m, \mathtt{cons}(n, \mathtt{nil}))\}) \\
&\qquad \longrightarrow_{inh} (G_3, \{\mathtt{cons}(n, \mathtt{split2}(\mathtt{nil})) < \mathtt{cons}(m, \mathtt{cons}(n, \mathtt{nil}))\}) \\
&\qquad \longrightarrow_{inh} (G_4, \{\mathtt{cons}(n, \mathtt{split2}(\mathtt{nil})) < \mathtt{cons}(m, \mathtt{cons}(n, \mathtt{nil}))\}) \\
&\qquad \longrightarrow_{inh} (G_5, \{\mathtt{cons}(n, \mathtt{nil}) < \mathtt{cons}(m, \mathtt{cons}(n, \mathtt{nil}))\})
\end{aligned}
$$

Now, we may easily classify the applications of $I_1, \ldots, I_5$ as well as the literals in $G_1, \ldots, G_5$ as contributing or non-contributing. For this, we consider the applications and goals bottom-up:

- I_5 is contributing as it does not generate any new subgoal. The fourth literal in G_1 and the third literal in $G_2, \ldots, G_5$ are contributing because they inherit to the principal literal for the application of I_5.
- I_4 is contributing as it modifies the third literal which is principal in the next contributing proof step I_5.

- I_3 and I_2 are non-contributing because they modify only the second resp. the first literal but these literals do not contribute to the subsequent proof, i.e. they do not inherit to literals that become principal in a contributing proof step.
- I_1 is contributing as it modifies all literals in the resulting subgoal, in particular, the third literal which becomes principal in the contributing proof steps I_4 and I_5. The principal literal for the application—the first one—contributes to the proof for G_1.

□

For adaptable inference systems, we may eliminate non-contributing proof steps from a performed proof, resulting in a pruned proof. With our new proof techniques for guiding proof search, we aim at avoiding non-contributing proof steps at all. In the following chapters, we present these applications in more detail.

Chapter 6

Guiding Proof Search with Forbidden and Mandatory Markings in Goals, and Obligatory and Generous Markings in Lemmas

In this chapter, we present a flexible framework for guiding proof search for adaptable inference systems (cf. Section 5.2). To simplify matters, we concentrate on one major task during the simplification process in (inductive) theorem proving: the application of conditional lemmas for rewriting or subsumption. In our case studies with QuodLibet, lemmas are applied in at least 50% of all proof steps. Due to its practical importance, this task has been studied for more than three decades, mostly under the label of "contextual rewriting".

For a deeper understanding of this chapter, it is necessary to recall the introduced notions for applying conditional lemmas such as head, condition, focus and context literals as well as definedness, condition, rewrite and order subgoals in Section 2.2.2.10 and 2.2.2.11; the partitioning of goals into principal part, cut-off part and context in Sections 5.1 and 5.2; and the notion of contribution in Section 5.3. Most of these notions are illustrated in Example 6.1 in Section 6.1. Roughly speaking, in clausal first-order logic, the goals to be proved and the lemmas that may be applied are given as clauses. A clause $\{\lambda_1, \ldots, \lambda_n\}$ may be interpreted as an implication $\overline{\lambda_1} \wedge \cdots \wedge \overline{\lambda_{n-1}} \Rightarrow \lambda_n$. We fix one literal in the lemma clause by calling it the *head literal*; the conjugates of the other literals are called *condition literals*. For each inference step, we also fix one literal in the goal clause, called *focus literal*; the conjugates of the other literals are called *context literals*. The conclusion of the lemma—its head literal—may be applied for proving the goal if the condition literals can be proved valid in the "context". According to [BM88a], we have to relieve the conditions.

In general, the process for applying a lemma can be divided into two steps: choosing a lemma; and checking the lemma for applicability and relieving its conditions. The first step can be supported by rippling techniques [BSvH$^+$93, Hut97]. The *relief test* during the second step has to be done by recursively calling the simplification process. We are interested in an *extensive* but *efficient* relief test. By "extensive", we mean that the test should not fail too often if the lemma application may contribute to the proof. Our flexible framework allows us to control the balance between extent and efficiency manually. Therefore, it is particularly suited for interactive theorem proving.

The elements in the goals that can be used during the relief test have to be restricted since the condition subgoals contain the original goal (cf. Section 2.2.2.10). Thus, without restrictions the relief test may result in an infinite process: the lemma can be applied to the condition subgoals over and over again. We concentrate on the question *which* elements can be used during the relief test.

Practically, there are two major ways to restrict proof steps that may be used during the relief test by marking elements in the goals:

1. In previous approaches known from the literature such as Contextual Rewriting in [Zha95] and Case Rewriting in [BR93], elements are excluded from certain condition subgoals. In the original approaches, excluded elements are completely eliminated from the subgoals resulting in *unsafe* applications, i.e. we may derive invalid goals by applying valid lemmas to valid goals (cf. Section 2.2.2.2). Within our safe inference system, we may model these approaches by marking excluded elements as *forbidden*. The meaning is that a forbidden element in a goal must not be principal in the application of an inference rule.

2. We propose a novel, alternative approach by marking elements in goals as *mandatory*: If we apply an inference rule to a goal, one of the mandatory elements must be principal. With a mandatory marking we may favor those proof steps that *locally* contribute to the proof.

By marking elements as mandatory instead of forbidden, we overcome some difficulties of previous approaches [Zha95, BR93]: As we can use every element during the relief test provided that there is also one mandatory element involved, we can achieve a more extensive relief test. Furthermore, we develop techniques to restrict the relief test in a user-defined way with *obligatory* and *generous* markings in the lemmas to achieve the right balance between efficiency and extent. Our flexible framework allows us to combine the different markings in an arbitrary way.

The influence of the markings in the goals on proof search can be defined in two steps:

1. we restrict the proof steps I that can be applied to a goal G according to the markings and the partitioning of goal G into principal part, cut-off part and context w.r.t. I;

2. we define the markings for the new subgoals.

Whereas we fix Step 1, the inheritance procedure in Step 2 may be realized in different ways. Step 2 may also be influenced manually. Therefore, we get a flexible mechanism to restrict proof search.

In Section 6.1, we present a simple example illustrating the advantages of our novel heuristics based on a mandatory marking in comparison to previous approaches based on a forbidden marking. In Section 6.2, we motivate, describe and illustrate our heuristics based on markings. For each heuristics, we identify proof patterns that cannot be handled with this heuristics. For our novel heuristics based on a mandatory marking, we can solve these problems at the expense of additional auxiliary lemmas or by using a generous marking which extends proof search. We compare the different heuristics in Section 6.3. There, we also provide evidence for the adequacy of our modeling of Contextual Rewriting with a forbidden marking. We conclude this chapter with some remarks in Section 6.4.

6.1 A Simple Example

The following example presents a proof pattern that can be handled by our novel heuristics but cannot be proved by previous approaches such as Contextual and Case Rewriting summarized in Section 6.2.2. It is taken from our case study that the greatest common divisor (gcd) of two natural numbers is idempotent, commutative and associative (at least if the numbers are unequal to zero).

Example 6.1 We consider an extension of the base specification from Section 4.2.1 consisting of sorts `Bool` (for boolean values with constructors `true` and `false`) and `Nat` (for natural numbers with constructors `0` and `s`) and defined operators `+`, `*`, `-`, `div`, `gcd`, `leq` and `div-p` which represent the corresponding arithmetic operations on natural numbers, a less-or-equal and a divisibility predicate on natural numbers. We consider the formal specification of `gcd` only, given by Axioms (6.1) to (6.4). The `gcd` of two natural numbers is defined if at least one of its arguments is unequal to zero. If exactly one of the arguments is unequal to zero this argument is the result of the operation. Otherwise, we recursively call `gcd` with the smaller argument and the difference of greater and smaller argument which ensures that the definition is terminating.

$$\begin{aligned}&\{\ \mathtt{gcd}(x,y)=x,\\ &\quad y\neq\mathtt{0},\\ &\quad x=\mathtt{0}\ \}\end{aligned}\tag{6.1}$$

$$\begin{aligned}&\{\ \mathtt{gcd}(x,y)=y,\\ &\quad x\neq\mathtt{0},\\ &\quad y=\mathtt{0}\ \}\end{aligned}\tag{6.2}$$

$$\begin{aligned}&\{\ \mathtt{gcd}(x,y)=\mathtt{gcd}(x,\mathtt{-}(y,x)),\\ &\quad \mathtt{leq}(x,y)\neq\mathtt{true},\\ &\quad x=\mathtt{0},\\ &\quad y=\mathtt{0}\ \}\end{aligned}\tag{6.3}$$

$$\begin{aligned}&\{\ \mathtt{gcd}(x,y)=\mathtt{gcd}(\mathtt{-}(x,y),y),\\ &\quad \mathtt{leq}(x,y)=\mathtt{true},\\ &\quad \neg\mathtt{def\ leq}(x,y),\\ &\quad x=\mathtt{0},\\ &\quad y=\mathtt{0}\ \}\end{aligned}\tag{6.4}$$

As auxiliary lemma for the associativity of `gcd`, we want to prove the following lemma on divisibility:

$$\begin{aligned}&\{\ \mathtt{div\text{-}p}(\mathtt{gcd}(x,y),z)=\mathtt{true},\\ &\quad \mathtt{div\text{-}p}(x,z)\neq\mathtt{true},\\ &\quad x=\mathtt{0}\ \}\end{aligned}\tag{6.5}$$

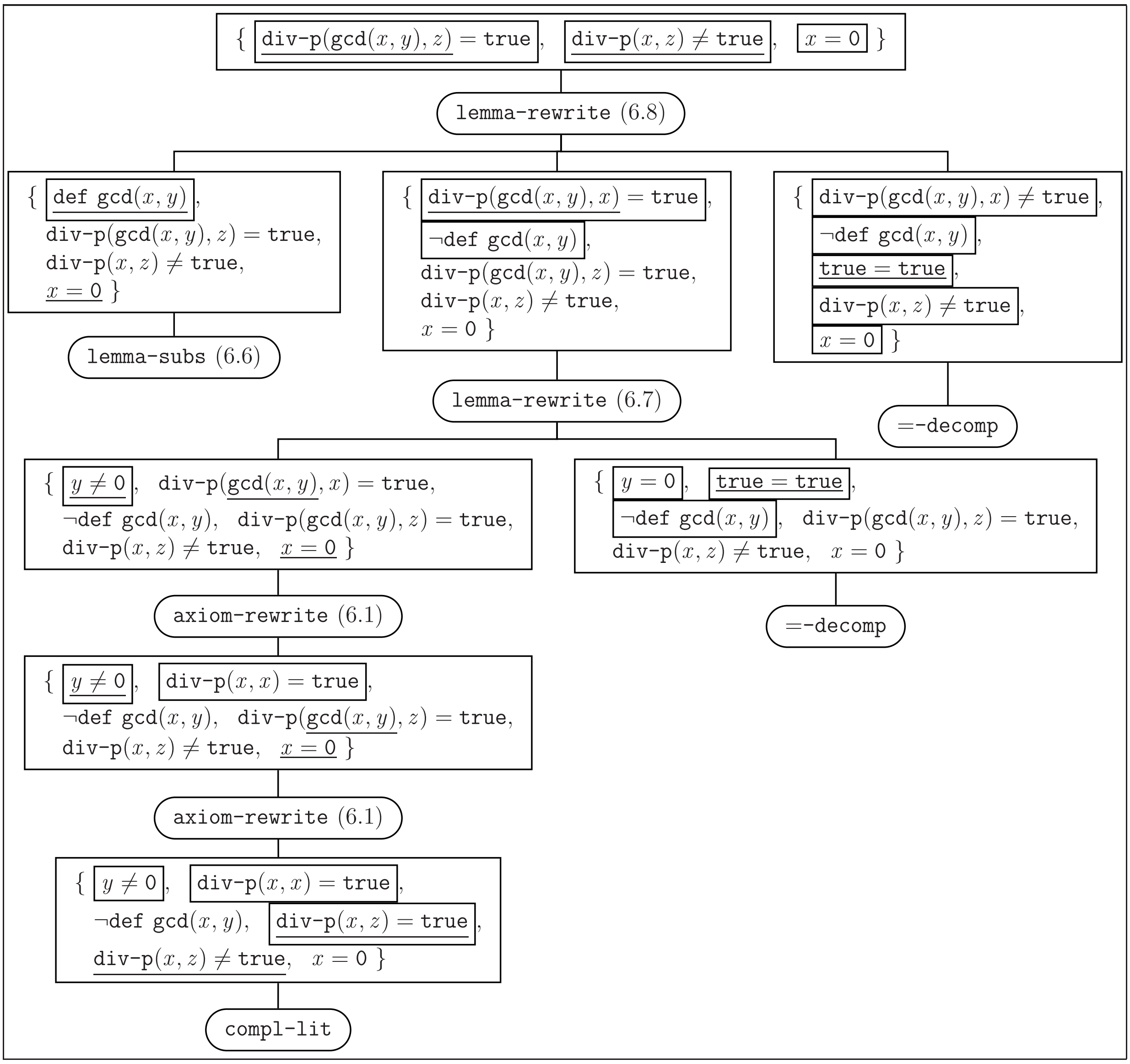

Figure 6.1: Proof State Tree for Goal (6.5) of Example 6.1

We assume that the following lemmas are activated for automatic applications:

$$\{\ \texttt{def gcd}(x, y),\ x = 0\ \} \tag{6.6}$$

$$\{\ \texttt{div-p}(\texttt{gcd}(x, y), x) = \texttt{true},\ y = 0\ \} \tag{6.7}$$

$$\{\ \texttt{div-p}(x, z) = \texttt{true},\ \texttt{div-p}(x, y) \neq \texttt{true},\ \texttt{div-p}(y, z) \neq \texttt{true}\ \} \tag{6.8}$$

For each of the axioms and lemmas, we choose the first literal as head literal for the following reasons:

- The axioms define operator `gcd` using the first literal as rewrite rule from left to right.

- Lemma (6.6) contains a definedness atom as first literal. Due to our monotonic semantics based on data models, we cannot prove negated definedness literals as focus literals. Therefore, the definedness atom should be present in the goal the lemma is applied to. Such a lemma is called a *domain* lemma as it establishes the domain of a (partial) operator.

- In Lemma (6.7), the left-hand side of the first literal is the only term that binds all variables of the lemma.

- In Lemma (6.8), the first literal is the positive literal in a Horn clause.

Figure 6.1 contains the whole proof state tree for Goal (6.5) as it is created by our novel heuristics (without regarding the weight component which is not relevant for the proof). The root goal node is rewritten by the conditional Lemma (6.8) using the substitution $[x \leftarrow \texttt{gcd}(x, y), z \leftarrow z, y \leftarrow x]$. The substitution can be determined by using the first literal of the root goal as focus literal and matching the head literal to the focus literal. The uninstantiated *extra* variable y can be bound by matching the third lemma literal to the second goal literal (cf. Section 6.2.5). Then, the third lemma literal is *directly fulfilled* by the second goal literal which itself is a *cut-off* literal. Thus, the first two goal literals are *principal* for the application. In Figure 6.1, principal literals are underlined and mandatory literals are framed. Our novel heuristics applies an inference rule automatically only if one of the principal literals is also mandatory, i.e. if one of the underlined literals is also framed. The application results in three new subgoals: one *definedness subgoal* (since the substitution binds a constructor variable to a non-constructor term), one *condition subgoal* and one *rewrite subgoal*. As there is a condition subgoal, the lemma is not *directly applicable*. The definedness subgoal is proved by a direct application of Lemma (6.6) for subsumption. Rewriting the condition subgoal with Lemma (6.7) leads to another condition subgoal. For its proof we rewrite the second and fourth literal with Axiom (6.1). Note that these literals have been the focus literals of the previous lemma applications that have generated the considered condition subgoal. Thus, these applications are possible only with our novel heuristics (cf. Section 6.2). Altogether, we get a *closed* proof state tree, i.e. a proof state tree whose leaves are inference nodes. Therefore, Lemma (6.5) is inductively valid provided that this holds true for the applied lemmas.

Analyzing the contribution of the proof steps, we observe that the only non-contributing proof step is the first application of Axiom (6.1) to rewrite the second literal. Indeed, this literal—the only new one—does not contribute to the proof for its subgoal. □

6.2 Flexible Control with Markings

Lemmas are provided to guide the proof process. On the one hand, they should be applied automatically as far as possible[1] to free the user from routine work. On the other hand, heuristics have to control the applications to guarantee the termination of the process within a reasonable amount of time. Thus, we have to find the right balance between extensiveness and efficiency.

[1]at least if they may contribute to the proof (cf. Section 5.3)

We essentially restrict proof search with markings in goals and lemmas (cf. Sections 6.2.1 to 6.2.4). To get a practicable method, we adapt additional heuristics known from the literature—in particular those presented in [BM88a]. We summarize these restrictions in Section 6.2.5.

Note that, in general, in our domain neither confluence nor termination properties can be assumed for rewriting with lemmas (cf. Example 6.20). Therefore, heuristics based on wellfounded orders are not always applicable. If applicable, these heuristics may be combined with our marking techniques.

6.2.1 Mandatory Markings in Goals

In this section, we present a novel heuristics based on a mandatory marking in goals. With this heuristics, we aim at avoiding non-contributing proof steps (cf. Section 5.3). The notion of contribution captures what we want but cannot be directly exploited for proof search: As contribution of a proof step depends on the proof performed, it can be *checked* only after the proof has been completed. But we can easily *ensure* that we perform only contributing proof steps by using one of the new elements as principal element in the *next* proof step.

Definition 6.2 (Locally Contributing Proof Steps)
Let $\mathcal{IS} = (\mathcal{I}, PG, SGs)$ be an inference system defined with principal part and context as in Definition 5.4. A proof step I in a proof P for a goal G *locally contributes* to P if every direct subgoal SG created by I contains a new element that becomes principal in the proof step performed for SG in P. □

Lemma 6.3 If every proof step in a proof P locally contributes to P, then every proof step contributes to P.

Proof. This lemma is proved easily by induction w.r.t. $\prec_P$ (cf. Section 5.2.1): Since P is a proof, every leaf is an inference node. Let G be an arbitrary goal node in P and $SI(G)$ the inference rule applied to G. If $nbSGs(SI(G)) = 0$, i.e. the applied inference rule does not generate any new subgoals, then proof step $SI(G)$ is contributing according to Definition 5.10. Otherwise, let SG be an arbitrary new subgoal generated by $SI(G)$. Since $SG \prec_P G$, the induction hypothesis is applicable to SG. Thus, the proof step $SI(SG)$ applied to SG contributes to P. Because every proof step locally contributes to P, one of the new elements in SG becomes principal w.r.t. $SI(SG)$ which is contributing. Since the inheritance relation defined in Definition 5.9 contains the reflexive closure, the new element in SG contributes to P. Therefore, proof step $SI(G)$ contributes to P. □

Note that, in general, a proof step does not have to be contributing even if it is locally contributing, because the new elements may become principal only in non-contributing proof steps.

As we will see, this strict usage of local contribution is too restrictive for guiding proof search. It excludes too many proofs in which all proof steps are contributing but some of them do not contribute locally. To be able to define local restrictions on proof steps in a flexible way, we introduce a mandatory marking in goals.

Restriction 6.4 (Caused by Mandatory Marking) An inference rule may be applied to a goal G with a *mandatory marking* only if one of the mandatory elements is principal in the proof step applied to G. □

If we mark only new elements in a subgoal as mandatory in a proof, it is ensured that all proof steps (locally) contribute to that proof. But then, the proof search is too restricted. It will find only "linear" proofs: We can apply only those inference rules that also use new elements introduced by the previous proof step. For a successful proof, however, it may be necessary to apply inference rules "in parallel" that are not linearizable. Such proofs are impossible with this strict usage of a mandatory marking as the following example illustrates.

Example 6.5 Given three boolean valued constants $\mathtt{p1}, \mathtt{p2}, \mathtt{p3}$, we assume the activation of Lemmas (6.9) and (6.10) and want to prove Goal (6.11)

$$\{\ \mathtt{p1} = \mathtt{p3}\ \} \tag{6.9}$$

$$\{\ \mathtt{p2} = \mathtt{p3}\ \} \tag{6.10}$$

$$\begin{aligned}\{\ &\mathtt{p1} = \mathtt{true}, \\ &\mathtt{p2} \neq \mathtt{true}\ \}\end{aligned} \tag{6.11}$$

To prevent trivial loops we use equations for rewriting just in one direction. We present our examples in such a way that equations are always applied for rewriting from left to right.[2] Therefore, the only way to prove Goal (6.11) is to rewrite p1 and p2 to p3. Then the resulting subgoal is tautological as it contains complementary literals. But if we mark only new elements as mandatory, this proof is prohibited since the second rewrite step does not use a new element. □

Alternatively, if all elements of subgoals are marked as mandatory, the marking has no effect and the search space contains too many proof steps that do not contribute to the proof. Our compromise results in the following default heuristics which can be fine-tuned with a *generous* marking in the lemmas as explained in Section 6.2.4:

Heuristics 6.6 (for Marking Elements as Mandatory) At the beginning of a proof attempt for a goal every element in the clause is marked as mandatory. Thus, there are no restrictions for performing proof steps.

For *applicability subgoals*—i.e. definedness or condition subgoals of applicative inference rules—the marking of the parent goal is not inherited to the subgoal, but a new set of mandatory elements is introduced that consists exactly of the new elements of the subgoal. With this *strict* marking heuristics, it is guaranteed that one of the new definedness or condition literals is used in the next proof step.

For order subgoals, we mark only the single new order atom as mandatory. In this case, the proof has to proceed by treating the order atom.

[2]Some lemmas such as the distributivity of multiplication over addition do not suggest a direction for rewriting themselves. In one situation it may be beneficial to apply them from left to right and in another situation vice versa. Therefore, the user may fix (during the activation of the lemma) the direction that is used for automatic applications.

For all other subgoals—i.e. rewrite subgoals or subgoals created by other inference rules—the mandatory elements of the parent goal stay mandatory in the subgoal (unless they are deleted) and are supplemented with all new elements of the subgoal. Thus, we use a *relaxed* marking heuristics. We can perform rewrite steps even if they do not contribute to the proof. This is helpful for the speculation of auxiliary lemmas. □

Example 6.7 (6.1 continued) In Figure 6.1, mandatory literals are framed, principal literals are underlined. Thus, we can apply an inference rule only if at least one of the underlined literals is also framed.

The proof starts at the root goal node with all literals marked as mandatory. After applying Lemma (6.8), the resulting definedness subgoal has one mandatory literal—the first one—that is handled by the following subsumption with Lemma (6.6). The mandatory literals of the condition subgoal—the second subgoal—are the first two literals. Note that the repeated application of Lemma (6.8) is prevented as none of its principal literals is mandatory anymore. Instead, the first literal is handled by the following rewrite step with Lemma (6.7), that introduces the first literal as the only mandatory literal for the new condition subgoal. This single mandatory literal is used in the rewrite step with Axiom (6.1). As this inference rule modifies the second literal of the resulting rewrite subgoal, it is added to the set of mandatory literals. Analogously, literal four is added to this set after the next rewrite step with Axiom (6.1). Finally, the inference rule `compl-lit` can be applied to the rewritten subgoal although not both literals are mandatory. It suffices that one mandatory literal is principal for the application. Note that all literals in the rewrite subgoal of the application of Lemma (6.8) are mandatory since the goal is not an applicability subgoal (cf. Heuristics 6.6). This is justified by the fact that an infinite loop of the same lemma application is already avoided because the original goal is not contained in the new subgoal. The relaxed mandatory markings heuristics for rewrite subgoals is, for instance, required for the second application of Axiom (6.1): Otherwise, $y \neq \mathtt{0}$ would not stay mandatory after the first rewrite step with Axiom (6.1) and the second application would not obey the restrictions caused by the mandatory marking. □

Example 6.8 As another example, we consider the defined operators `less` and `+`, given by the following axioms:

$$\{\ \mathtt{less}(\mathtt{0}, \mathtt{s}(y)) = \mathtt{true}\ \} \tag{6.12}$$
$$\{\ \mathtt{less}(x, \mathtt{0}) = \mathtt{false}\ \} \tag{6.13}$$
$$\{\ \mathtt{less}(\mathtt{s}(x), \mathtt{s}(y)) = \mathtt{less}(x, y)\ \} \tag{6.14}$$

$$\{\ \mathtt{+}(x, \mathtt{0}) = x\ \} \tag{6.15}$$
$$\{\ \mathtt{+}(x, \mathtt{s}(y)) = \mathtt{s}(\mathtt{+}(x, y))\ \} \tag{6.16}$$

Given the additional lemmas

$$\{\ \mathtt{def}\ \mathtt{+}(x, y)\ \} \tag{6.17}$$
$$\begin{aligned}\{\ &\mathtt{less}(x, \mathtt{+}(x, y)) = \mathtt{true},\\ &y = \mathtt{0}\ \}\end{aligned} \tag{6.18}$$
$$\begin{aligned}\{\ &\mathtt{less}(x, z) = \mathtt{true},\\ &\mathtt{less}(x, y) \neq \mathtt{true},\\ &\mathtt{less}(y, z) \neq \mathtt{true}\ \}\end{aligned} \tag{6.19}$$

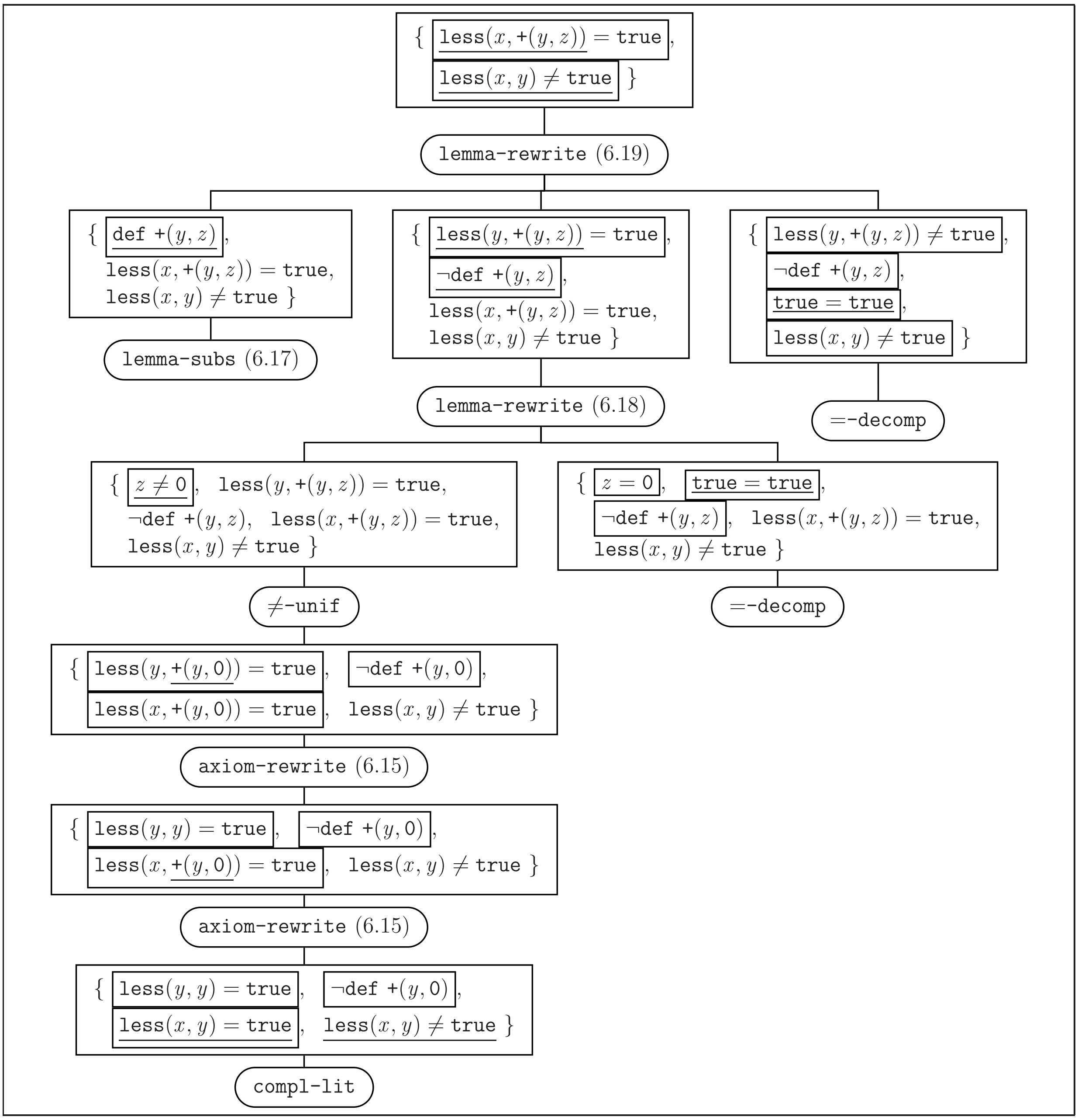

Figure 6.2: Proof State Tree for Goal (6.20) of Example 6.8

we want to prove the inductive validity of the following goal by simplification:

$$\{\ \texttt{less}(x, \texttt{+}(y, z)) = \texttt{true}, \quad \texttt{less}(x, y) \neq \texttt{true}\ \} \tag{6.20}$$

For each of the axioms and lemmas, we choose the first literal as head literal for the following reasons: The axioms define operators `less` and `+` using the first literal as rewrite rule from left to right. In Lemma (6.18), the left-hand side of the first literal is the only term that binds all variables of the lemma. In Lemma (6.19), the first literal is the positive literal in a Horn clause.

Figure 6.2 contains the whole proof state tree for Goal (6.20) as it is created by our novel heuristics with a mandatory marking. Again, mandatory literals are framed, principal literals are underlined.

The proof starts at the root goal node with all literals marked as mandatory. The root goal node is rewritten by the conditional Lemma (6.19) using the substitution $[z \leftarrow \texttt{+}(y, z)]$. The substitution can be determined by using the first literal of the root goal as focus literal and matching the head literal to the focus literal. The second lemma literal is directly fulfilled by the second goal literal (binding the extra variable y to itself). The application results in three new subgoals (from left to right): one definedness subgoal, one condition subgoal and one rewrite subgoal. The definedness subgoal has one mandatory literal—the first one—that is handled by the following subsumption with Lemma (6.17). The mandatory literals of the condition subgoal are the first two literals. Note that these mandatory literals prevent the repeated application of Lemma (6.19). Instead, the first literal is handled by the following rewrite step with Lemma (6.18), that introduces the first literal as the only mandatory literal for the new condition subgoal. This single mandatory literal is used by the inference rule $\neq$`-unif`. As this inference rule modifies the first three literals of the resulting subgoal, they become the mandatory literals. The following rewrite steps do not alter the sets of mandatory literals as we do not start new sets for rewrite goals. This results in one rewrite step that does not contribute to the proof. Finally, the inference rule `compl-lit` can be applied to the rewritten subgoal although not both literals are mandatory. It suffices that one mandatory literal is principal for the application. □

Examples 6.1 and 6.8 contain a basic proof pattern that cannot be proved with Contextual Rewriting (cf. Section 6.2.2.1) but with our novel heuristics. This proof pattern is illustrated in Example 6.9 in an abstract way. We use it for comparing our novel heuristics (cf. Figure 6.3a described in Example 6.9) with Contextual Rewriting (cf. Figure 6.3b described in Example 6.14) and Case Rewriting according to [BR93] (cf. Figure 6.3c described in Example 6.16).

Example 6.9 ([Zha95], simplified) Given three boolean valued constants `q1`, `q2`, `q3`, we assume the activation of the following lemmas

$$\begin{aligned}\{\ &\texttt{q1} = \texttt{true},\\ &\texttt{q2} \neq \texttt{true}\ \}\end{aligned} \tag{6.21}$$

$$\begin{aligned}\{\ &\texttt{q1} = \texttt{true},\\ &\texttt{q3} \neq \texttt{true}\ \}\end{aligned} \tag{6.22}$$

$$\begin{aligned}\{\ &\texttt{q2} = \texttt{true},\\ &\texttt{q3} = \texttt{true}\ \}\end{aligned} \tag{6.23}$$

and want to prove the goal

$$\{\ \texttt{q1} = \texttt{true}\ \} \tag{6.24}$$

As Lemmas (6.21) and (6.22) are Horn clauses we use the first literal as head literal. Lemma (6.23) does not suggest a head literal itself. We may use an arbitrary one or both literals. Due to efficiency considerations and as the lemmas are symmetric in `q2` and `q3`, we decide to choose just the first one.

Using a mandatory marking, the proof is found automatically (cf. Figure 6.3a). In the condition subgoal after applying Lemma (6.23), literal `q1` $=$ `true` can be used as focus

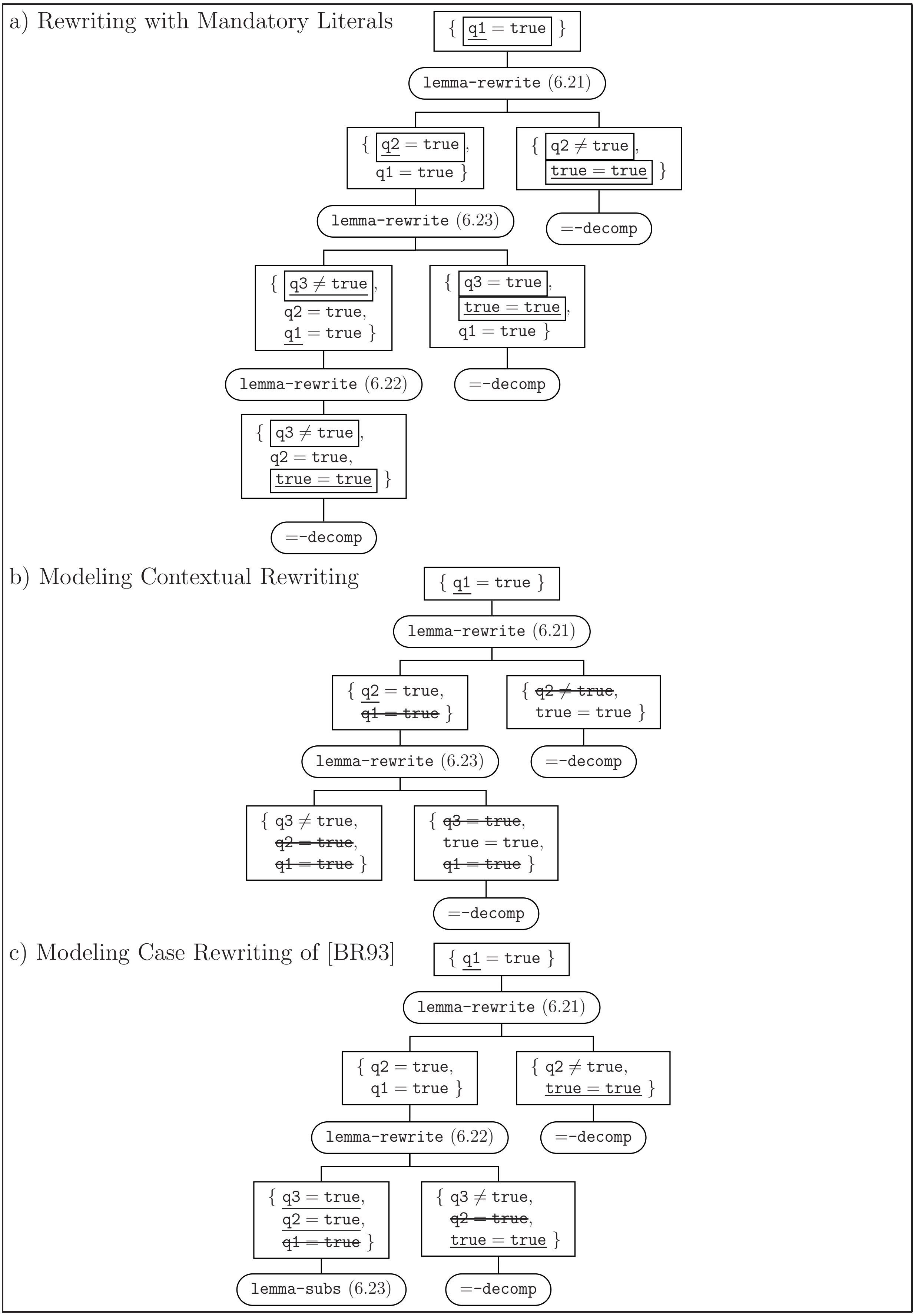

Figure 6.3: Proof State Trees for Example 6.9

literal to rewrite `q1` to `true` although this literal is not mandatory. This can be done since the condition literal of the applied lemma is mandatory. □

For an extensive relief test, the following property would be useful: If a goal can be proved by simplification without any restrictions on the elements that can be used then it can also be proved obeying the restrictions caused by a mandatory marking. Unfortunately, this strong property does not hold as will be shown in Example 6.10, a simple generalization of Example 6.9.

The interaction of head literals in lemma clauses and mandatory literals in goal clauses restricts the search space of the simplification process very much: In most cases, a lemma will be applied to a goal only if the head literal of the lemma is mandatory in the goal. The proof step then transfers the mandatory marking from the head literal to its condition literals. This direction can be inverted automatically only if the gap between the head literal and one of the condition literals can be closed in one step within the goal clause, i.e. if one condition literal is a mandatory literal of the goal clause as in Example 6.9. Otherwise, we have to use auxiliary lemmas to bridge the gap.

Example 6.10 Given five boolean valued constants $\texttt{r1}, \ldots, \texttt{r5}$, we assume the activation of the following lemmas

$$\{\, \texttt{r1} = \texttt{true}, \; \texttt{r2} \neq \texttt{true} \,\} \tag{6.25}$$

$$\{\, \texttt{r1} = \texttt{true}, \; \texttt{r3} \neq \texttt{true} \,\} \tag{6.26}$$

$$\{\, \texttt{r2} = \texttt{true}, \; \texttt{r4} \neq \texttt{true} \,\} \tag{6.27}$$

$$\{\, \texttt{r3} = \texttt{true}, \; \texttt{r5} \neq \texttt{true} \,\} \tag{6.28}$$

$$\{\, \texttt{r4} = \texttt{true}, \; \texttt{r5} = \texttt{true} \,\} \tag{6.29}$$

and want to prove the goal

$$\{\, \texttt{r1} = \texttt{true} \,\} \tag{6.30}$$

The specification is illustrated in Figure 6.4 by solid lines. We assume that the first literal is used as head literal for each lemma. There is a gap of two steps e.g. between $\texttt{r1} = \texttt{true}$ and $\texttt{r5} = \texttt{true}$ that cannot be closed automatically. Using the mandatory markings heuristics, our automatic proof control performs two proof attempts which are

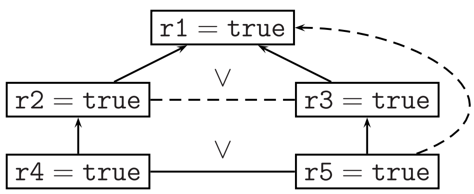

Figure 6.4: Illustration of Example 6.10

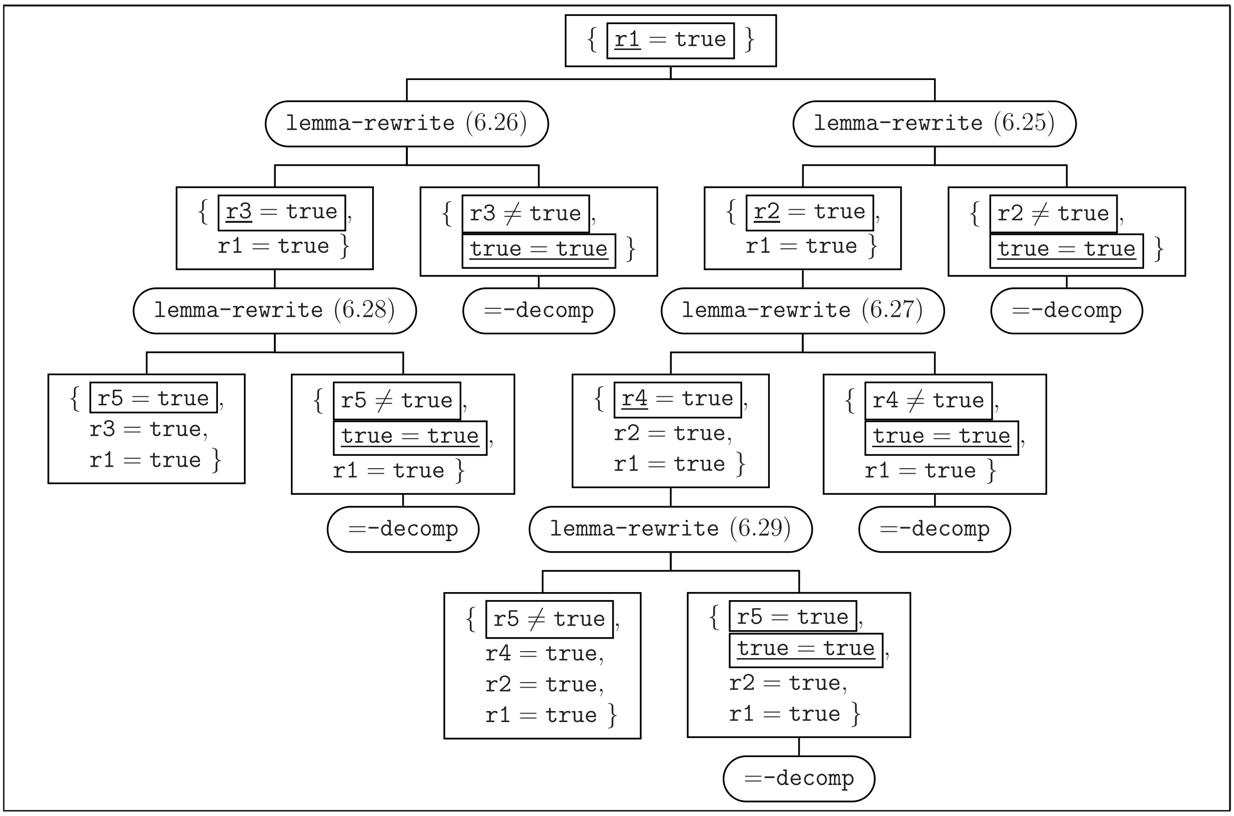

Figure 6.5: Two Failed Proof Attempts for Goal (6.30) of Example 6.10

illustrated in Figure 6.5. None of the lemmas can be applied to one of the two open goals without violating the restrictions caused by the mandatory marking. In the open goal of the first proof attempt, for instance, only literal $\mathtt{r5} = \mathtt{true}$ is marked as mandatory. Therefore, the only way to obey the restriction caused by the mandatory marking would be to apply Lemma (6.29). But as `r4` is not present in the goal, we cannot apply the lemma for rewriting `r4` as required by the activation. The same argument holds true for the open goal of the second proof attempt, Lemma (6.28) and operator `r3` which is not present in the goal.

We can overcome this situation by introducing e.g. one of the following auxiliary lemmas (illustrated in Figure 6.4 by dashed lines):

$$\{\, \mathtt{r2} = \mathtt{true}, \quad \mathtt{r3} = \mathtt{true} \,\} \tag{6.31}$$

$$\{\, \mathtt{r1} = \mathtt{true}, \quad \mathtt{r5} \neq \mathtt{true} \,\} \tag{6.32}$$

Each of these lemmas as well as Goal (6.30) (after activating one of (6.31), (6.32)) can be proved automatically. Goal (6.30), for instance, can be proved analogously to Goal (6.24) in Example 6.9 if we activate Lemma (6.31) (cf. Figure 6.3a replacing $\mathtt{q}i$ with $\mathtt{r}i$). If we activate Lemma (6.32) we can prove the open goal of the second proof attempt in Figure 6.5. Thus, we can bridge the gap. □

Theorem 6.11 states that we can always close gaps with auxiliary lemmas.

Theorem 6.11 If a goal can be proved by simplification without any restrictions on the elements that can be used then it can also be proved with a mandatory marking with the help of some auxiliary lemmas which themselves can be proved with a mandatory marking. More precisely, if a proof violates the restrictions at n goal nodes then we need at most n auxiliary lemmas.

Proof. A proof step that creates a subgoal that does not possess any new literal cannot contribute to a proof. Hence, it can be eliminated from the proof. Thus, we can assume that a proof contains at least one new literal in each subgoal. As we mark at least one of the new literals as mandatory, each subgoal contains at least one mandatory literal. If a proof step applied to a goal G in the proof violates the restrictions caused by a mandatory marking we can introduce a new lemma consisting of this goal G. As each proof attempt for a new lemma starts with all literals marked as mandatory, the proof of the lemma succeeds with a mandatory marking. Whatever head literal is chosen for this lemma, it can be applied to prove goal G which formerly violated the restrictions caused by a mandatory marking. The lemma is applicable because at least one literal in G is mandatory. □

Unfortunately, the required auxiliary lemmas cannot be calculated automatically. In fact, the automatic generation of lemmas according to the last proof would counteract the mandatory marking because proof search would continue for the auxiliary lemmas with all elements marked as mandatory again. Nevertheless, the auxiliary lemmas may be manually extracted from failed proof attempts. In contrast to this, Contextual Rewriting may not even be able to make use of auxiliary lemmas simply because one cannot build a bridge when a bank is forbidden.

6.2.2 Forbidden Markings in Goals

Other approaches known from the literature perform the relief test in such a way that certain elements are excluded from the generated subgoals. In these approaches, excluded elements are completely eliminated from the subgoals, resulting in unsafe lemma applications. Within our flexible framework, we model these approaches in a safe way by introducing a forbidden marking in goals. In this section, we consider adaptable inference systems defined with cut-off part (cf. Definition 5.6).

Restriction 6.12 (Caused by Forbidden Marking) An inference rule may be applied to a goal G with a *forbidden marking* only if all forbidden elements that are principal in this proof step are cut-off elements. □

We account for the elimination of forbidden elements in the approaches known from the literature in the following way: Once an element is marked as forbidden in a goal G, it remains forbidden in the whole proof attempt for G. Our modeling with a forbidden marking improves the versions in the literature insofar as forbidden elements may serve as cut-off elements. Let G be a goal with a forbidden marking and G' be the goal derived from G by eliminating all forbidden elements. According to Definition 5.6, an inference rule I' is applicable to a goal G' iff the inference rule $adaptI(I', G)$ is applicable to G. Therefore, proof search is essentially the same in both cases except that the approaches that eliminate forbidden elements have to prove

- additional subgoals that are cut off by the forbidden elements;
- stronger goals since conditions in form of forbidden elements are missing. This may even cause a failure of a proof attempt because of a subgoal which is, in fact, trivial if we do not eliminate forbidden elements.

Thus, we consider the use of a forbidden marking as an adequate and safe alternative for modeling previous approaches from the literature. The additional cut-off elements do not change the search space, but relax the success criterion of our proof search. In Section 6.3, we perform case studies to validate the adequacy of our modeling of Contextual Rewriting and to demonstrate the additional benefits of using forbidden elements as cut-off elements.

6.2.2.1 Contextual Rewriting

Contextual Rewriting in the narrower sense is used e.g. in `NQTHM` [BM88a], `ACL2` [KMM00], `RRL` [Zha95], and more recently in `RDL` [AR03]. These approaches vastly differ in their simplification process e.g. in the way they use equality information. `NQTHM` [BM88a] and `ACL2` [KMM00] use the cross fertilization technique while `RRL` [Zha95] uses a constant congruence closure algorithm. In `RDL` [AR03], decision procedures can be used by the simplification process. Nevertheless, they use the same literals to perform the relief test: The focus literal in the applicability subgoals as well as all downfolded literals are marked as forbidden. On the one hand, Contextual Rewriting is not very restrictive because it admits non-contributing proof steps. On the other hand, it is often too restrictive as can be seen in our examples:

Example 6.13 (6.7 continued) The second application of Axiom (6.1) in Figure 6.1 rewrites a literal initially used as focus literal. Thus, Contextual Rewriting fails. □

Example 6.14 (6.9 continued) Two lemmas have to be applied to the same goal literal to perform a successful proof. But after applying one lemma, the focus literal is forbidden for the rest of the proof attempt. This situation is depicted for one proof attempt in Figure 6.3b where forbidden literals are marked by crossing them out. The proof attempt fails at the left-most leaf as $\mathtt{q1} = \mathtt{true}$ cannot be used anymore. It is not possible to overcome this situation with auxiliary lemmas. □

Not surprisingly, Example 6.10—a generalization of Example 6.9—cannot be proved with Contextual Rewriting either. Nevertheless, a slight modification changes the example in such a way that it can be proved with Contextual Rewriting but not with our novel heuristics with a mandatory marking (in the simple form presented in Section 6.2.1 and without auxiliary lemmas).

Example 6.15 Given six boolean valued constants $\mathtt{s1}, \ldots, \mathtt{s6}$, we assume the activation of the following lemmas

$$\begin{aligned}\{\ & \mathtt{s1} = \mathtt{true}, \\ & \mathtt{s3} \neq \mathtt{true}\ \}\end{aligned} \tag{6.33}$$

$$\begin{aligned}\{\ & \mathtt{s2} = \mathtt{true}, \\ & \mathtt{s4} \neq \mathtt{true}\ \}\end{aligned} \tag{6.34}$$

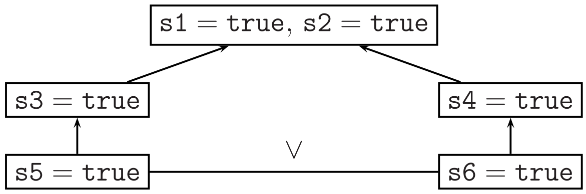

Figure 6.6: Illustration of Example 6.15

$$\{\ \texttt{s3} = \texttt{true},\ \texttt{s5} \neq \texttt{true}\ \} \tag{6.35}$$

$$\{\ \texttt{s4} = \texttt{true},\ \texttt{s6} \neq \texttt{true}\ \} \tag{6.36}$$

$$\{\ \texttt{s5} = \texttt{true},\ \texttt{s6} = \texttt{true}\ \} \tag{6.37}$$

and want to prove the goal

$$\{\ \texttt{s1} = \texttt{true},\ \texttt{s2} = \texttt{true}\ \} \tag{6.38}$$

The specification is illustrated in Figure 6.6. We assume that the first literal is used as head literal for each lemma. Whereas Contextual Rewriting can apply Lemmas (6.33) to (6.37) one after the other, our heuristics based on mandatory markings cannot close the gap without auxiliary lemmas or generous literals (cf. Sections 6.2.1 and 6.2.4). □

6.2.2.2 Case Rewriting

Case Rewriting tries to overcome the difficulties of Contextual Rewriting by a special treatment of lemmas that can be applied to rewrite the same redex alternatively, such as e.g. Lemmas (6.21) and (6.22) in Example 6.9. In this sense, our approach with a mandatory marking is a novel form of Case Rewriting. There are at least two further approaches known from the literature [BR93, KR90].

The approach for Case Rewriting proposed in [KR90] restricts the relief test by order constraints which we cannot use in general for our application domain as we allow non-terminating operator definitions.

In the approach of Case Rewriting in [BR93], a term is rewritten by a set of n lemmas resulting in $n+1$ new subgoals: For each lemma one rewrite subgoal is created; additionally one *well-coveredness* subgoal is produced. This last subgoal is to guarantee the completeness of the case split w.r.t. the given lemmas.

Example 6.16 (6.9 continued) For the specification of Example 6.9, this Case Rewriting approach can be modeled by applying the two lemmas in succession as depicted in Figure 6.3c. In the well-coveredness goal—i.e. the left-most goal—only the condition literals may be used. As the case split for Lemmas (6.21) and (6.22) is complete according to Lemma (6.23), the proof can be completed. □

The approach of [BR93] seems to have the following limitations: The set of lemmas applied depends only on the focus literal but not on the context. A single well-coveredness goal is sufficient only if all lemmas differ only in a single condition literal. The well-coveredness goal cannot be proved if the case split is incomplete.

Example 6.17 (6.13 continued) The applications of Lemma (6.8) and Axiom (6.1) in Figure 6.1 do not form a complete case split. Actually, they even do not rewrite the same redex. Thus, Case Rewriting according to [BR93] cannot be applied. Moreover, in contrast to [BR93], our approach with a mandatory marking can make use of other goal literals to prove the well-coveredness subgoal. □

6.2.3 Obligatory Markings in Lemmas

On the one hand, the use of a mandatory marking as explained in Section 6.2.1 results in an *extensive* relief test. But as we call the simplification process recursively for any condition subgoal whose condition literal is not directly fulfilled in the goal (cf. Section 2.2.2.10), it may be very time-consuming. On the other hand, using only directly applicable lemmas is a very *efficient* but not extensive relief test because it checks only syntactic equality. As a compromise, we introduce an obligatory marking in lemmas that restricts the relief test for obligatory elements to the efficient syntactic test. This guides the proof search in a user-defined way, manually controlling the degree of extent and efficiency for each lemma separately.

Restriction 6.18 (Caused by Obligatory Marking)
A lemma with an *obligatory marking* may be applied to a goal G only if all obligatory elements are directly fulfilled in the goal. □

Thus, a lemma with an obligatory marking is only applicable if the instantiated obligatory elements are present in the goal. Therefore, we may interpret the obligatory marking as a user-defined means to extend the principal part in the goal w.r.t. the applied lemma.

By marking elements as obligatory, we restrict proof search. In doing so, we may prevent the automatic derivation of proofs. Thus, elements are *automatically* marked as obligatory only if the head literal of the lemma is an equation that is used as rewrite rule with a *general term* as left-hand side, i.e. a term of the form $f(x_1, \ldots, x_n)$ in which the x_i are pairwise different variables. Without obligatory elements, the relief test would be invoked too often in this case, most of the time unsuccessfully.

Heuristics 6.19 (for Marking Elements as Obligatory)
Our automatic default heuristics chooses one obligatory element[3] if the lemma is used as rewrite rule with a general term as left-hand side. □

[3]More precisely, the next literal after the head literal is marked as obligatory. Therefore, the user may influence this heuristics with the order of the literals in the lemma clause. Certainly, this default heuristics may also be completely overwritten manually.

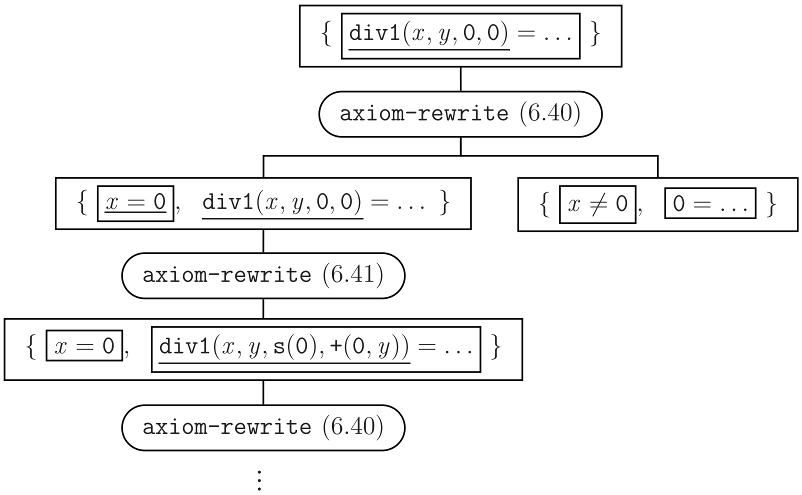

Figure 6.7: Non-terminating Relief Test in Example 6.21

Example 6.20 If we use the first literal of the trichotomy of less

$$\begin{aligned} \{\ & \mathtt{less}(x, y) = \mathtt{true}, \\ & \mathtt{less}(y, x) = \mathtt{true}, \\ & x = y\ \} \end{aligned} \tag{6.39}$$

as head literal then it is activated as rewrite rule for operator `less` with a general term as left-hand side. When applying this lemma, the relief test reduces the question whether x is less than y to the question whether y is not less than x and whether they are unequal. But this relief test is in most cases not simpler than the original problem. In Example 6.8, the activation of Lemma (6.39) would increase the number of inference steps from 9 to 12. But if the context says that neither x is less than y nor y is less than x then we can derive that x is equal to y by Lemma (6.39). Thus, we may prove the remaining literals of the clause in the possibly very useful context that x is equal to y. Therefore, the automatic application of the lemma should be restricted by marking the second lemma literal as obligatory. □

Example 6.21 Another example is given by the axioms of a division operator:

$$\begin{aligned} \{\ & \mathtt{div1}(x, y, u, v) = u, \\ & x \neq v\ \} \end{aligned} \tag{6.40}$$

$$\begin{aligned} \{\ & \mathtt{div1}(x, y, u, v) = \mathtt{div1}(x, y, \mathtt{s}(u), \mathtt{+}(v, y)), \\ & x = v\ \} \end{aligned} \tag{6.41}$$

Without marking literals as obligatory the relief test illustrated in Figure 6.7 does not terminate. During the relief test for Axiom (6.40), Axiom (6.41) can be applied for rewriting because the mandatory literal directly fulfills the condition literal of Axiom (6.41). Since the rewriting changes the second goal literal it becomes mandatory and thus can be used for further rewrite steps with the axioms. If the conditions of the axioms are marked as obligatory, already the first relief test is prevented. □

6.2.4 Generous Markings in Lemmas

The efficiency of the relief test can be improved and manually controlled with obligatory markings. To enhance the extent of the relief test based on a mandatory marking in a user-defined way, we now introduce generous markings in lemmas.

The idea of our mandatory markings is to prefer (locally) contributing proof steps, and, therefore, to prevent non-contributing proof steps. But as explained in Example 6.5, applying only those proof steps that locally contribute to a proof, restricts proof search too much and prevents too many proofs where all proof steps are contributing but some of them do not locally contribute to the proof. Therefore, our default Heuristics 6.6 for marking mandatory elements in subgoals applies *strict* or *relaxed* mandatory marking heuristics depending on the type of the subgoal, i.e. whether it is an applicability subgoal, an order subgoal or another subgoal. But even with this default Heuristics 6.6, proof search is restricted in such a way that we may require auxiliary lemmas just to compensate for the restrictions caused by our mandatory markings (cf. Example 6.10 and Theorem 6.11). Auxiliary lemmas may be required for two reasons:

1. to find a proof at all (cf. Example 6.10); or

2. to improve the efficiency of proof search by introducing shortcuts in the search space. With auxiliary lemmas, proof search may be guided on a fine-grained level. But this burdens the user with having to pick suitable auxiliary lemmas.

Instead of introducing auxiliary lemmas that just compensate for the restrictions caused by mandatory markings, we may vary the mandatory markings heuristics. On the one hand, if we use only the strict mandatory markings heuristics, all proof steps locally contribute to the proof. On the other hand, if we use only the relaxed mandatory markings heuristics, we do not pose any restrictions on proof search at all. As a compromise, we introduce generous markings of lemma literals. With generous elements the default behavior for marking mandatory elements in the subgoals as described in Heuristics 6.6 can be changed in a flexible way.

Restriction 6.22 (Caused by Generous Marking) If a lemma with a *generous marking* is applied to a goal G, it causes the following restriction on the mandatory marking of a condition or rewrite subgoal SG:

If SG is generated from a lemma literal which is marked as generous, then the mandatory marking of SG is inherited from G and supplemented with all the new elements of SG. Thus, the marking is generated with the relaxed marking heuristics used for rewrite subgoals in Heuristics 6.6.

If the corresponding lemma literal is not generous, then a new set of mandatory elements is introduced for SG consisting exactly of the new elements in SG. In this case, the marking is generated with the strict marking heuristics used for condition subgoals in Heuristics 6.6. □

Heuristics 6.23 (for Marking Elements as Generous) Our automatic default heuristics marks exactly the head literal of every rewrite lemma as generous. □

Note that, with Heuristics 6.23, the mandatory marking Heuristics 6.6 in Section 6.2.1 now works exactly as without a generous marking before.

Generous elements relax the restrictions caused by a mandatory marking. Therefore, we may avoid some auxiliary lemmas.

Example 6.24 (6.10 continued) If all literals in Lemmas (6.25) and (6.26) are generous, we may first apply these two lemmas. This results in the following clause:

$$\{\ \texttt{r2} = \texttt{true},\ \texttt{r3} = \texttt{true},\ \texttt{r1} = \texttt{true}\ \} \tag{6.42}$$

Due to the generous marking, all literals in this clause are mandatory. Therefore, we can prove this clause in the same way as Lemma (6.31). □

Using the relaxed mandatory markings heuristics for generous elements clearly extends the search space. Therefore, one would expect that we get a less efficient relief test. Often, this holds true but, in general, it is not that easy to analyze the effects of generous elements on proof search. Generous elements enable more proof steps that do not locally contribute to the proof. For these proof steps, we do not know whether they contribute to the proof. Often, they are non-contributing. But, sometimes, they may enable additional proof steps that introduce shortcuts during proof search. They may avoid many failed proof attempts resulting in improved efficiency. Thus, generous elements may have similar effects on proof search as auxiliary lemmas: They may enable a proof at all and they may increase the efficiency of proof search. But they also extend the search space—just as auxiliary lemmas do—which may decrease the efficiency of proof search as well.

In general, the introduction of auxiliary lemmas allows us to guide proof search on a more fine-grained level than this can be done by marking elements as generous. In the former case, we may introduce just the lemma instance required for closing the proof state tree, whereas, in the latter case, many lemmas may become applicable—most of them resulting in unsuccessful proof attempts. Therefore, auxiliary lemmas do not extend the search space as much as generous markings do. Thus, for efficiency reasons it is advantageous to use auxiliary lemmas. But generous markings relieve the user of the burden of picking these auxiliary lemmas. Thus, we recommend their use in a limited way for complicated proofs. This is done for the case study about LPO in Chapter 8.

Often, the coarse-grained extension of the search space caused by generous elements introduces too many non-contributing proof steps which contain unnecessary proof obligations in terms of open goal nodes. These additional proof obligations countervail the benefits of the generous markings in such a way that, actually, the efficiency of proof search decreases when using generous markings on their own. In Chapter 7, we introduce *upward propagation*—a reuse technique that prunes a proof by eliminating all non-contributing proof steps. The combination of generous markings and upward propagation allows us

1. to search for a proof performing proof steps that do not locally contribute; and

2. to eliminate proof obligations of non-contributing proof steps.

Only in this combination, generous markings can display their full power. Therefore, we do not consider generous markings in this chapter anymore. Instead, we validate the benefits of generous markings in Section 8.2.2.3 on the basis of our case study about the LPO after having introduced upward propagation in Chapter 7.

Even in combination with upward propagation, generous markings should be used with caution. We recommend their use if a lemma (or a literal in a goal) is expected to be essential for the proof, i.e. a proof cannot be found without using the lemma (or the literal in the goal), and the proof itself is expected to be easy but not necessarily linear. Usually, we assume that these properties hold true for goals containing definedness atoms which are proved by applying corresponding domain lemmas. Therefore, we usually mark as generous:

- negated definedness atoms in all lemmas because they generate definedness atoms in the corresponding condition subgoals; and
- all literals in domain lemmas.

Furthermore, if an operator f is defined in terms of other function symbols using defining rules which are not recursive, we may decide to reduce terms containing operator f with its defining rules in any case. Then, the literals in the defining rules of operator f should be marked as generous.

6.2.5 Further Heuristics for Guiding Proof Search

To get a practicable method for guiding proof search we apply further heuristics:

- We prevent repeated applications of the same inference rule with the same principal literals within one proof attempt (disregarding cut-off literals): For the application of an inference rule I to a goal G, we inspect all the applications on the branch of the proof state tree from the root goal to G.

  In particular, when using generous markings for condition literals in lemmas, this mechanism is required for avoiding trivial rewrite loops: As the condition subgoal generated from a generous condition literal inherits the mandatory marking from its parent goal, the same inference step is applicable again.

- We do not apply lemmas that apparently do not support the proof of the goal. There may be two reasons for this: Firstly, the conditions of the lemma and the context of the goal are inconsistent, i.e. the context contains the negation of one condition. Secondly, the focus literal of the goal is rewritten to an obviously unsatisfiable literal as e.g. $t \neq t$.

- During an automatic proof attempt, we do not want to guess any instantiations of lemma variables. Thus, *extra* variables—i.e. variables that are not bound by matching the head literal to the focus literal—must be instantiated by matching condition literals to context literals.

- *Permutative* lemmas as e.g. the commutativity of + are applied only w.r.t. a fixed wellfounded total term order. By this, we hope to prevent infinite rewrite chains with permutative lemmas.

- To prevent infinite loops when proving applicability subgoals, the maximal recursion depth can be restricted.

6.3 Case Studies

In this chapter, we have presented novel heuristics to restrict the relief test for conditional lemmas. To validate our novel heuristics on some real case studies, we have to integrate them into an inductive proof process. As in Section 4.3.1 we want to study solely the effects on proof search caused by the different heuristics based on markings. Therefore, we compare the different heuristics with as few differences as possible. Instead of comparing different systems that implement the various heuristics, we use the same proof process as well as the same specifications within a single system. Thus, we realize Contextual Rewriting as explained in Section 6.2.2.1. At the end of this section we will also point out how the results obtained by our simulations carry over to other provers.

Due to its flexible inference system (cf. Sections 2.2.2 and 4.2.3), we choose our inductive theorem prover QuodLibet to perform the simulations. The inference rules for rewriting and subsumption allow for the simulation of the different heuristics easily. Note that systems based on Contextual Rewriting eliminate the focus literal from the condition subgoals. Thus, their underlying inference systems would have to be changed to simulate our novel heuristics.

Since our heuristics are particularly suited for the application of lemmas but do not provide special means for the application of linear arithmetic we use the simpler waterfall of Section 3.2.2 for the comparison. Thus, we use Configuration (A) from Section 4.3.1 which triggers more applications of lemmas. In the following, we present further details about the automatic application of lemmas during the phases `reduce1` and `reduce2` of the waterfall. As explained in Section 3.1.4, lemmas may be applied automatically only if they are *activated*. During the activation of a lemma the user may provide the head and the obligatory and generous literals of the lemma. Otherwise, they are determined by some heuristics (cf. Section 6.2.3 for obligatory, Section 6.2.4 for generous, and [SS04] for head literals). Lemmas are tested in reverse activation order, which may be changed to influence the proof search. If a head literal is an equation (whose left-hand side is not a variable), it is used for *rewriting*; otherwise, for *subsumption*. Subsumption is checked for first; then the subterms of the focus literal are tested for rewriting, using an innermost left-to-right strategy. If a lemma can be applied and all its applicability subgoals can be proved, its application will not be deleted anymore. Thus, no alternative proof attempts for successful applications will be tried out during this tactic execution. Contrariwise, a lemma application—together with all proof attempts of the applicability subgoals—is deleted if the relief test fails. This results in a backtracking step. Further details can be found in [SS04].

For a fair evaluation of forbidden markings, we have slightly modified the automatic application of axioms during our simplification process when using a forbidden marking: If axioms can be applied to the same redex alternatively (such as the axioms of the `gcd` in Example 6.1) a Cut with the condition literal(s) will be performed automatically using inference rule `lit-add`. This then enables the application of all axioms. Otherwise, already the second application would be prevented because the rewrite literal becomes forbidden after the first application. This simulates the *operator unfolding* operation in systems like

Example	Lemmas	Man. Interact.	Autom. Appl.	Del.	Fin. P.	Runtime
`sqrt` (E)	38	2 + 0	529	2	527	1.46
`Lpo`	154	5 + 67	5458	404	5054	36.89

Table 6.1: Complexity of the Additional Case Studies

`NQTHM` where all axioms are given in one operator definition. Together with the additional admission of forbidden literals as cut-off literals, this results in a modeling where Contextual Rewriting can display its full power.

Beside the case studies `sortalgos`, `gcd`, `exp-exhelp`, and `sqrt` (H) from Section 4.3.1, we use the following additional case studies for the comparison:

`sqrt` (E): This example contains another proof of the irrationality of $\sqrt{2}$ based on ideas of Euclid of Alexandria.

`Lpo`: This is a simplified version of our case study about the lexicographic path order LPO (cf. Chapter 8). In this version, we prove that every LPO is a simplification order. Furthermore, we prove the equivalence of two different implementations of the LPO [Löc04].

Table 6.1 illustrates the complexity of the additional case studies. It contains the number of lemmas (constant for all heuristics), and, for our novel heuristics with a mandatory and obligatory marking, the number of manual interactions (manually applied inference rules + manually chosen induction order), the number of automatically applied inference rules (including the later deleted ones), the number of deleted inference rules due to a failed relief test, the number of inference rules in the final proof and the runtime in seconds measured by a CMU Common Lisp system on a machine with a 1 GHz Intel III processor and 4 GB RAM. For the complexity of the other case studies, we refer to the entries of Configuration (A) in Table 4.1.

Table 6.2 contains for each example and each heuristics based on a combination of **o**bligatory, **m**andatory and **f**orbidden markings the following statistics: in column "Open Lemmas", the number p of proof state trees that cannot be closed with this heuristics; the entries in the other columns take into account only those proof state trees that are closed with *all* heuristics: We do not count applications and deletions of inference steps of open proof state trees because failed proof attempts tend to create large proof state trees, tampering our results.

From the statistics in Table 6.2, we draw the following conclusions:

- Since the specification of `gcd` contains non-terminating rewrite rules, it can be performed only with an *obligatory* marking. For the other examples, obligatory markings restrict the search space without influencing the resulting proofs very much: The number of inference steps in the final proof is nearly the same regardless of the usage of obligatory markings.
- The *best heuristics w.r.t. efficiency* (as underlined in the table) use a combination of mandatory/obligatory **{m,o}** and forbidden/obligatory **{f,o}** markings, respectively.

Heur.	Open Lemmas	Autom. Appl.	Del.	Fin. P.	Runtime
Example `sortalgos`					
∅	2	2348	143	2205	6.77
{o}	0	2143	19	2124	5.56
{m}	0	2072	107	1965	5.53
{m,o}	0	2007	40	1967	5.09
{f}	0	2354	234	2120	6.31
{f,o}	0	2168	60	2108	5.13
Example `gcd`					
∅	—	—	—	—	—
{o}	0	911	5	906	2.13
{m}	—	—	—	—	—
{m,o}	0	893	9	884	2.14
{f}	—	—	—	—	—
{f,o}	**9**	974	18	956	2.07
Example `exp-exhelp`					
∅	0	3290	9	3281	299.28
{o}	0	3290	9	3281	298.90
{m}	0	1278	116	1162	7.26
{m,o}	0	1278	116	1162	7.22
{f}	0	1342	204	1138	7.85
{f,o}	0	1342	204	1138	7.77
Example `sqrt` (H)					
∅	1	2008	969	1039	16.11
{o}	0	1059	26	1033	5.98
{m}	0	1064	31	1033	6.22
{m,o}	0	1046	13	1033	5.85
{f}	0	1021	60	961	5.40
{f,o}	0	980	25	955	5.28
Example `sqrt` (E)					
∅	0	477	4	473	1.24
{o}	0	471	0	471	1.18
{m}	0	477	4	473	1.24
{m,o}	0	471	0	471	1.25
{f}	3	483	16	467	1.27
{f,o}	**3**	467	0	467	1.26
Example `Lpo`					
∅	1	11125	1089	10036	291.37
{o}	0	7621	848	6773	153.85
{m}	0	5746	971	4775	46.44
{m,o}	0	4988	330	4658	31.28
{f}	1	32946	26667	6279	370.79
{f,o}	**1**	8050	2135	5915	50.89

Table 6.2: Comparison of the Different Heuristics

As the same simplification process is used, these two heuristics can differ only if a proof step cannot be applied due to the restrictions caused by mandatory or forbidden markings. Only for the `Lpo` example, a major advantage in efficiency can be determined, favoring our novel **{m,o}** heuristics.

- As printed in bold in the table, of 466 lemmas in total *13 lemmas cannot be proved* with **{f,o}**, but our novel **{m,o}** heuristics proves all of them.

Note that in our modeling of Contextual Rewriting with **{f,o}**, 13528 subgoals are created, but 1296 definedness and 648 condition subgoals—as well as the proof trees rooted in them—are cut-off by using forbidden literals as cut-off literals.

Finally, to answer the question about the adequacy of our simulation of Contextual Rewriting, we converted one of our case studies into a proof script for a prover based on Contextual Rewriting. We chose the `gcd` example as it contains most failed proof attempts with a forbidden marking. As prover we used `NQTHM` [BM88a] because we did not want to use the decision procedures for linear arithmetics integrated in `ACL2`. Instead, we used the *shell* principle to define our own type for natural numbers. We applied the following transformations to the original proof script: The specification style is changed from constructor to destructor recursion. Partial definitions are simulated using `F` as undefined value. As `NQTHM` is untyped, we explicitly restrict all lemmas to natural numbers only. These transformations were quite easy. Additionally, we added one operator definition just to provide a suitable induction scheme for the proof of one lemma as well as four auxiliary lemmas to enable the proof of two lemmas—namely Lemma (6.7) of Example 6.1 and a similar lemma that are proved in QuodLibet by mutual induction. These are two of the nine lemmas that failed with our simulation of **{f,o}** in QuodLibet. From the remaining seven lemmas, only two are not proved automatically. Note that this is not a weakness of our simulation of Contextual Rewriting. Instead, the difference is caused by different induction principles: `NQTHM` uses explicit induction (cf. Section 3.1.2). Thus, it does not apply lemmas inductively but splits, at the beginning of a proof, the induction steps of conditional lemmas in different cases and immediately adds a promising induction hypothesis. In contrast to this, we use descente infinie (cf. Section 3.1.3). Therefore, we have to use the relief test more often in QuodLibet. Beside the failed proofs in the statistics, two lemmas proved by our novel heuristics with simplification are proved by induction in `NQTHM`. Thus, these proofs are more complex in `NQTHM`.

6.4 Concluding Remarks

Rewriting with conditional lemmas is at the heart of many (inductive) theorem provers. Especially for interactive theorem provers, it is essential not only to prove as many lemmas automatically as possible but also to restrict proof search in a suitable way such that the proof process stops within a reasonable amount of time.

In this chapter, we have developed a framework that allows us to restrict proof search in a flexible way using heuristics based on markings in goals and lemmas. Within our framework we can simulate Case Rewriting and Contextual Rewriting with a forbidden marking in goals. The adequacy of our simulation of Contextual Rewriting is demonstrated by carrying over a case study to `NQTHM`. Furthermore, we have developed a novel heuristics **{m,o}** based on the orthogonal concepts of a mandatory marking in the goals and an obligatory marking in the lemmas. For the comparison of the heuristics we chose the well established application domain of rewrite-based simplification in inductive theorem proving. Our simulation of Contextual Rewriting **{f,o}** is competitive with our novel heuristics **{m,o}** regarding efficiency but not regarding extent. Nevertheless, the benefits of our novel heuristics are slightly decreased in theorem provers using explicit induction because they do not perform a relief test for induction hypotheses.

Neither Case Rewriting nor Contextual Rewriting nor our novel heuristics are perfect. For all of them, we have identified proof patterns that cannot be handled with the basic version of these heuristics. For our novel heuristics, we can always overcome these difficulties using auxiliary lemmas or a generous marking in the lemmas relaxing the restrictions caused by a mandatory marking. This is not possible e.g. for Contextual Rewriting. Furthermore, our framework allows us to choose between the different heuristics and to combine them easily. With obligatory and generous markings in lemmas we can fine-tune the degree of extent and efficiency of the proof search manually.

Our framework depends only on the partitioning of goals into principal part, cut-off part and context according to the inference rule applied. The basic distinction between principal part and context was already introduced in Gentzen's seminal work on sequent calculi [Gen35]. Therefore, this partitioning (and also the refinement with cut-off formulas) should pose no problems for inference systems based on sequent calculi. Indeed, a similar form of lemma application occurs in all practice-oriented mathematical assistance systems and the concepts behind our marking as mandatory, forbidden, obligatory, and generous are all in great demand and applicable, provided that we extend the inheritance procedures to the new inference rules in a meaningful way. As explained in Section 6.3, systems based on Contextual Rewriting eliminate the focus literal from the condition subgoals. Thus, their underlying inference systems have to be changed as a prerequisite for the integration of our marking techniques. This may require significant technical effort. Then, the marking techniques may be realized using wrapper functions for the inference rules as it is done in QUODLIBET.

Chapter 7

Reusing Contributing Proofs

Primarily, the automatic proof control *searches* for a proof by applying inference rules to goals based on heuristics. This proof search is often very time-consuming. Since we do not have perfect heuristics, we cannot decide a priori whether a proof step will contribute to a proof, is superfluous, or—even worse—prevents a successful proof. In the latter case, we have to backtrack to revise a former proof step. Contrariwise, if we can *reuse* an old proof, we are able to perform only contributing proof steps without any backtracking. Therefore, reuse is much more efficient than proof search. But the reuse capabilities are often rather limited. In this chapter, we try to enhance these capabilities for adaptable inference systems (cf. Section 5.2).

Usually, *arbitrary* proofs are searched for. *Minimal* proofs are not considered unless proofs are presented to human users or proof checkers. This seems to be reasonable because finding a minimal proof for a clause Γ may be more difficult than finding arbitrary proofs at all. Nevertheless, minimal proofs may indicate that some of the premises in Γ are not needed for the validity of Γ. The elimination of these unused elements delivers a stronger theorem with higher potential for reuse. This is particularly useful for lemmas that have been speculated automatically.

In this chapter, we refine our notion of *contribution* (cf. Section 5.3). More precisely, we present algorithmic definitions of *essentially contributing proof steps* and *essentially contributing elements* in a goal which mutually depend on each other. Instead of *searching* for minimal proofs, we analyze and *prune* each proof found w.r.t. its (essential) contribution.[1] We derive a pruned proof by deleting each non-contributing proof step. Every non-contributing element of goal G for proof P is superfluous for the validity of G. Therefore, the notion of contribution extracts the essence of a proof.

We do not apply this pruning only for the whole proof of the root goal but for every subgoal. We may reuse the proof for a goal if the following *reuse property* holds:

Proof P for goal G is *reusable* for goal G'
if G' contains all contributing elements of G for P. (RP)

[1]In this chapter, we usually write "contribution" instead of "(essential) contribution". The proposed reuse techniques may be applied with "contribution" as defined in Definition 5.10 as well as with "essential contribution" as defined in Definition 7.4. Note that "essential contribution" entails "contribution". Thus, more non-contributing proof steps may be identified and eliminated if we apply Definition 7.4.

Reuse can be exploited in two different ways: Firstly, we can check property (RP) directly. This is sensible if we transfer a proof *sidewards* in the proof state tree. Secondly, we can propagate a proof for a subgoal *upwards* in the proof state tree as long as on the path upwards none of its contributing elements has been introduced by the parent inference rule of the subgoal, i.e. the parent goal contains all contributing elements of the subgoal. This deletes proof steps that turned out to be superfluous and automatically improves the proof in a restricted way. The deletion of the proof steps may also remove open proof obligations.

Sideward reuse may be interpreted as a restricted form of lemma application. Note that we do not allow for any modifications to the contributing elements in property (RP) as e.g. instantiation with a substitution σ. Therefore, we can check this property very efficiently. The usefulness of our reuse mechanism depends on the fact that similar subgoals with the same contributing elements will be generated within one proof. Our case studies provide evidence for this.

In Section 7.1, we illustrate our reuse mechanisms for adaptable inference systems with an example on an abstract level. In the subsequent sections, we present our approach in more detail: We refine the notion of contribution in Section 7.2. Upward propagation and sideward reuse as well as their integration into a simple waterfall model are described in Section 7.3. In Section 7.4, we validate our approach with some case studies and conclude this chapter with a survey of related work in Section 7.5.

7.1 A Motivating Example

In this section, we illustrate our reuse mechanisms on an abstract level. Figure 7.1 contains a proof state tree for an abstract adaptable inference system $\mathcal{IS}$ with inference rules I_j (cf. Section 5.2). As usual, goals are illustrated by their clauses given in rectangular boxes. For reference purposes, we display a unique number for each goal in the lower right corner enumerating goals in the tree in preorder. A clause is represented by the literals λ_i it contains. The applied inference rules are illustrated in rounded boxes. For reference purposes, we annotate each inference rule with an index that corresponds to the goal it is applied to. Note that the tree contains the open goals G_{10} and G_{12}. Thus, the clause in the root goal G_1 is not proved. Our approach restructures the given proof state tree automatically:

- It propagates a pruned proof for G_3 upwards in the proof state tree to prove G_2. In doing so, the open goal G_{10} is eliminated.
- It reuses another pruned proof for goal G_4 to prove goal G_{12}.

Therefore, this restructuring results in a closed proof state tree. Furthermore, our approach identifies literal λ_4 as superfluous. Figure 7.2 displays the pruned proof state tree for this strengthened clause. The labeling of the goals in this proof state tree refers to the corresponding goals in the original proof state tree—e.g. goals $G_{9'}$ and $G_{9''}$ in Figure 7.2 are derived from goal G_9 in Figure 7.1. The same holds true for inference rules.

To perform the restructuring of the proof, we do not have to know the concrete inference rules. Instead, we need information only about the partitioning of the goals into principal part and context w.r.t. the applied inference rules, and about the elements that are modified

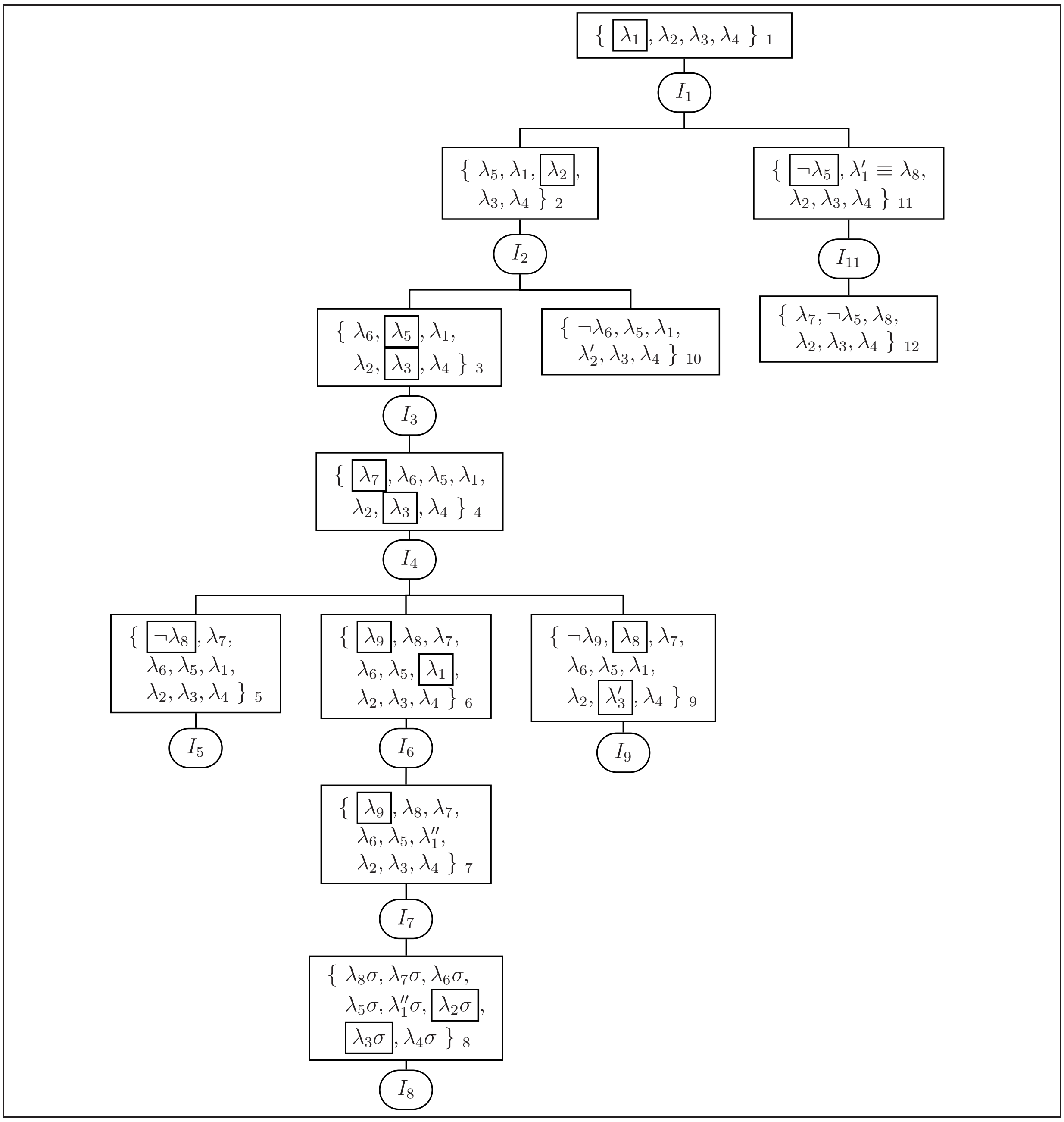

Figure 7.1: Proof State Tree with Two Open Goals

in the new subgoals w.r.t. the parent goal. This information is given for adaptable inference systems with functions *princ*, *context*, *side*, *non-side* and *nside-src* (cf. Section 5.2). In Figures 7.1 and 7.2, principal literals are framed; the other literals form the context. If a literal is derived from a principal literal in the parent goal, we use a primed version of this principal literal in the subgoal. For new literals we use new indices. But this distinction is not essential. In G_{11}, literal λ'_1 is derived from λ_1 in G_1 whereas $\neg\lambda_5$ is a new literal. We assume that λ'_1 happens to be the same as λ_8 indicated by $\lambda'_1 \equiv \lambda_8$ in G_{11}. For all inference rules in Figure 7.1 except for I_7, *non-side* and *nside-src* are identity functions, i.e. the context is transferred to the subgoals unmodified. The remaining literals in the subgoals are generated by function *side*. In G_8, the principal literal λ_9 of G_7 is removed.

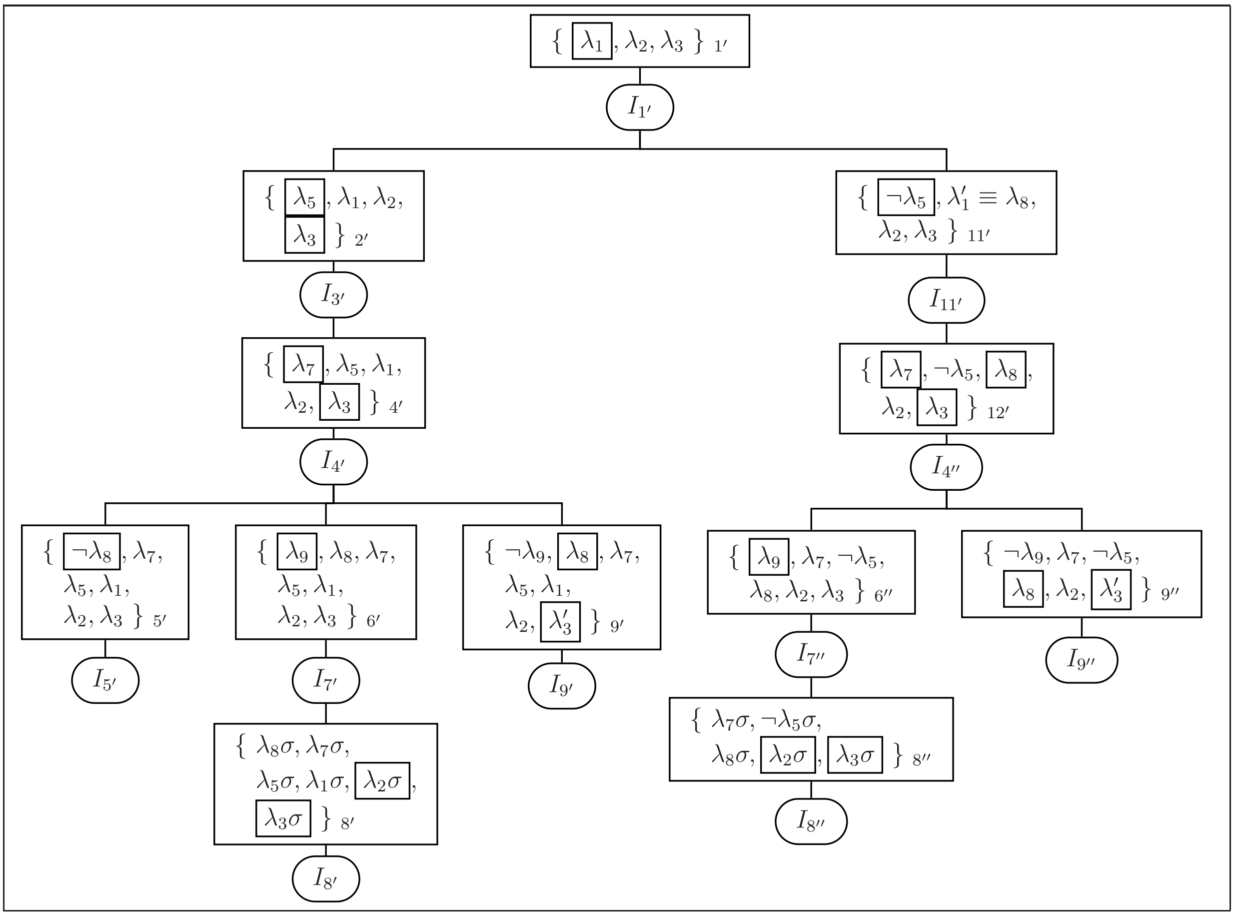

Figure 7.2: Proof State Tree Closed with Upward Propagation and Sideward Reuse

All other literals in G_8 are modified in a uniform way applying substitution σ. This is done by function *non-side*. In this case, for a set Γ' of literals in G_8 generated from *context*(I_7), function *nside-src* may return an arbitrary set $\Gamma \subseteq$ *context*(I_7) such that $\Gamma\sigma = \Gamma'$.

Note that the proof state tree is not generated arbitrarily. For each abstract inference rule, we can find a concrete inference rule of QuodLibet that uses the same principal elements and performs the same modifications to derive the new subgoals from the parent goal. Consider, for instance, I_4 in Figure 7.1. Assume that we are given a conditional rewrite lemma $\{ \neg\lambda_9, \lambda_8, \lambda_7, s = t \}$, and that $s = t$ may be used to rewrite λ_3 to λ'_3. For the application of the lemma, we have to guarantee that the conditions of the lemma are fulfilled. Therefore, inference rule `lemma-rewrite` generates a case split for every condition that is not directly present in G_4 resulting in condition subgoals G_5 and G_6. The rewrite subgoal G_9 establishes all conditions of the lemma. Thus, we can replace λ_3 with λ'_3 in G_9. Note that all other literals in G_4 are transferred to subgoals G_5, G_6 and G_9 without any modifications. In this example, literals λ_3 and λ_7 are principal: Literal λ_3 is rewritten in G_9, literal λ_7 prevents the creation of one condition subgoal. As a second example, consider inference rule I_7. It corresponds to $\neq$`-unif` in QuodLibet.

For adaptable inference systems, an inference rule I with parent goal G can be adapted to goal G' if only G' contains the principal part of G (for I). More precisely, function *adaptI* returns an adapted inference rule I' that can be applied to G'. For each new subgoal of I',

Goal	Contributing Elements	Contributing Proof Steps
G_5	$\neg\lambda_8$	I_5
G_8	$\lambda_2\sigma, \lambda_3\sigma$	I_8
G_7	$\lambda_9, \lambda_2, \lambda_3$	I_7
G_6	$\lambda_9, \lambda_2, \lambda_3$	I_7
G_9	λ_8, λ_3'	I_9
G_4	$\lambda_7, \lambda_2, \lambda_3$	I_4
G_3	$\lambda_5, \lambda_2, \lambda_3$	I_3

Table 7.1: Contribution for the Proof of G_3 in Figure 7.1

function *selectAG* selects a subgoal of I that generates the same elements for the common parts of G and G'. In Figure 7.2, we can adapt inference rule $I_{4'}$ to $G_{12'}$ because $G_{12'}$ contains the principal literals λ_3 and λ_7 of $G_{4'}$. Note that the adapted inference rule $I_{4''}$ applied to $G_{12'}$ generates one subgoal less than the application of $I_{4'}$ to $G_{4'}$. We assume that λ_8 is used as cut-off literal in $G_{12'}$. It cuts off the subgoal that corresponds to $G_{5'}$. As cut-off literals are considered to be principal, λ_8 is framed in $G_{12'}$. Furthermore, λ_8 in goals $G_{6''}$ and $G_{9''}$ is not newly created but inherited from $G_{12'}$. The literals common in $G_{12'}$ and $G_{4'}$, however, generate the same literals in the corresponding subgoals.

How do we find out whether a whole proof P for G can be reused for G' in our setting for adaptable inference system? We can adapt the first proof step of P if G' contains at least the principal elements of G. There will be created at most as many new subgoals for G' as for G. If we can reuse the proofs of the subgoals of G for those of G' we are done. For this, we have to guarantee that the subgoals of G' contain the principal elements of the corresponding subgoals of G. Furthermore, we have to guarantee this property recursively for all offspring goals.

With function *nside-src* we can determine the source in the parent goal of those elements in a subgoal SG that are required for the proof of SG. Thus, we can precompute in a bottom-up style all the elements in G that are required for P.[2] We call these elements *contributing elements* of G for P. During their computation we may recognize that some proof steps are not really needed for the proof: If a goal G contains all the contributing elements of one of its subgoals SG, we can reuse the proof of SG for G eliminating the proof step originally applied to G. The remaining proof steps are called *contributing proof steps*. The computations of contributing elements and proof steps mutually depend on each other. The computations are performed bottom-up in the proof state tree.

Table 7.1 contains the contributing elements and contributing proof steps for the proof of G_3 in Figure 7.1. The contributing literals of G_5, G_8 and G_9 are the principal literals. In G_7, literals λ_2 and λ_3 are contributing as they are responsible for generating the contributing literals in G_8. Proof step I_6 is not contributing since the contributing literals of G_7 are present in G_6. Thus, for computing the contributing elements of G_6, the principal literal λ_1 in G_6 w.r.t. I_6 is not added to the contributing elements of G_7. Eventually, we get λ_5, λ_2 and λ_3 as contributing literals of G_3 which are present in G_2. Therefore, we can propagate the proof of G_3 upwards to G_2. Note that this implies that we need not find a proof for G_{10}. This shows that pruning a proof may save time since unnecessary proof obligations are

[2]This computation refines the notion of contribution as presented in Definition 5.10 (cf. Section 5.3).

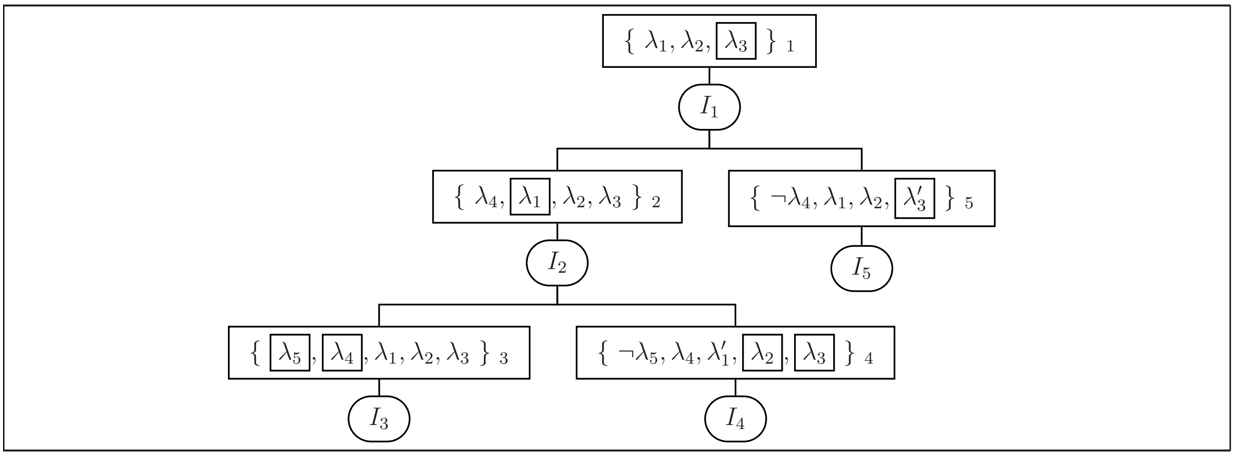

Figure 7.3: Illustration of Problems with Preliminary Definition of Contribution

detected. Since the contributing literals of G_4 are present in G_{12}, we can reuse its *pruned* proof to close the whole proof state tree as presented in Figure 7.2. Note that we cannot reuse the *original* proof of G_4 for G_{12} because the non-contributing proof step I_6 requires literal λ_1 which is not present in G_{12}.

7.2 Essential Contribution

In this section, we present a refined definition of contribution in comparison to Definition 5.10. This definition is technically more involved. But it may identify more non-contributing proof steps. Therefore, more proof steps may be eliminated by pruning the proof state tree.

If a proof contains at least one non-contributing proof step, Definition 5.10 may classify too many elements and proof steps as contributing although they are not relevant for the proof.

This problem is illustrated in Figure 7.3. According to Definition 5.10, proof steps I_3, I_4 and I_5 are contributing because they do not generate any new subgoals. Therefore, literals λ_5, λ_4 are contributing for G_3, literals λ_2, λ_3 for G_4, and literal λ'_3 for G_5. Proof step I_2 is non-contributing since none of the new literals $\neg\lambda_5$ and λ'_1 in its second subgoal G_4 becomes principal in the proof of this subgoal. But according to the preliminary definition, literals $\lambda_4, \lambda_2, \lambda_3$ contribute to the proof of G_2 as they become principal in contributing proof steps. Therefore, proof step I_1 is also contributing because in its first subgoal G_2 the new literal λ_4 contributes to the proof of this subgoal, and in its second subgoal G_5 the new literal λ'_3 contributes to the proof of this subgoal.

Since I_2 is non-contributing because of its second subgoal, we may reuse its proof for G_2, i.e. we may prove G_2 with I_4. Therefore, we should not consider the contributing elements λ_5, λ_4 of G_3 as contributing for G_2 but only the contributing elements λ_2, λ_3 of G_4. Then, I_1 is non-contributing because in the proof of its first subgoal G_2 the new literal λ_4 does not contribute to the proof.

```
1  I ← SI(G);
2  if nbSGs(I) = 0 then
3      contribI(G) ← I;
4      contrib(G) ← princ(I);
5  else
6      foreach SG ∈ SGs(I) do
7          calc-contrib(SG);
8      𝒢 ← {SG ∈ SGs(I) | contrib(SG) ⊆ G};
9      if 𝒢 ≠ ∅ then
10         SG ← selectCG(𝒢);
11         contribI(G) ← contribI(SG);
12         contrib(G) ← contrib(SG);
13     else
14         contribI(G) ← I;
           contrib(G) ← princ(I)+
15                 ⋃_{i=1}^{nbSGs(I)} nside-src(I, i, contrib(SG_i(I)) − side(I, i)) ;
```

Figure 7.4: **Procedure** *calc-contrib*(G)

For each non-contributing proof step I applied to goal G, there exists a set $\mathcal{G}$ of subgoals that are responsible for the non-contribution of I, i.e., for each subgoal $SG \in \mathcal{G}$, none of the new literals in SG contributes to the proof of SG. In the definition of *essential contribution* (cf. Definition 7.4), for each non-contributing proof step I applied to goal G, we transfer the contributing elements of just one of the subgoals $SG \in \mathcal{G}$ to G. The contributing proof steps and elements resulting from the proofs of other subgoals are not considered anymore. Note that the uniqueness of the definition of essential contribution depends on a heuristics that chooses exactly one of the subgoals in $\mathcal{G}$. We model this heuristics with function *selectCG*. CG stands for "contributing goal".

Let P be a proof for G, and $I = SI(G)$ the proof step performed for G (cf. Section 5.2.1). If G' is a goal with $princ(I) \subseteq G'$, we can apply $I' = adaptI(I, G')$ to G'. If we can recursively apply this adaptation for every new subgoal $SG_j(I')$ using $SG_i(I)$ with $i = selectAG(I, I', SG_j(I'))$, then we can reuse the whole proof.

We aim at defining a *contributing* part $contrib(G)$ of G in such a way that (a pruned version of) proof P for G can be applied to G' if $contrib(G) \subseteq G'$. During the analysis of proof P, we may recognize proof steps that are superfluous, i.e. the contributing part of a subgoal is present in one of its ancestor goals. We use this information to extract from P a *pruned* proof P'. This pruned proof is identified by storing in $contribI(G)$ the *contributing inference rule* that can be adapted to G.

Procedure *calc-contrib* calculates $contrib(SG)$ and $contribI(SG)$ for each offspring SG of a proved goal G in a bottom-up style (cf. Figure 7.4). $contrib(SG)$ and $contribI(SG)$ are stored in global hashes as a side-effect of the procedure.

- If G is proved with a single inference rule, i.e. no new subgoals are created, $princ(I)$ defines the contributing part, and the inference rule is contributing (Lines 2–4).

- Otherwise, we first calculate the contribution of each subgoal (Lines 6–7). If the contributing part of at least one subgoal is present in goal G we select one such subgoal SG with function *selectCG* and transfer its contributing part and inference rule to G (Lines 9–12). Otherwise, the inference rule applied to G is contributing. Beside $princ(I)$ we have to adapt the contributing elements of each subgoal that are generated by $context(I)$ (Lines 13–15). This is done using function *nside-src*.

Function *selectCG* in Line 10 provides a heuristics for selecting a subgoal whose contribution is propagated upwards in the proof state tree. In our current implementation, we prefer those subgoals whose contributing part consists of as few elements as possible. Thereby, we hope to increase the probability that we can further propagate the proof upwards in the proof state tree. From these subgoals again, we choose one with as few proof steps as possible to get shorter proofs.

Obviously, we can improve procedure *calc-contrib* as we do not have to recalculate the contribution for a goal SG if we call *calc-contrib* for its ancestor G. Instead, we can just use the hashed values $contrib(SG)$ and $contribI(SG)$. We may also use procedure *calc-contrib* for proof state trees with multiple proof attempts. In this case, we just have to iterate over all inference rules applied.

For inference systems defined with principal part and context as in Definition 5.4, procedure *calc-contrib* (cf. Figure 7.4) has the following properties which can be proved by induction w.r.t. $\prec_P$ (cf. Section 5.2.1):

Lemma 7.1 Let $\mathcal{IS}$ be an inference system defined with principal part and context as in Definition 5.4. Then, procedure *calc-contrib* is terminating (cf. Figure 7.4).

Proof. We have to prove that each call *calc-contrib*(G) is terminating. We perform induction on goal G w.r.t. $\prec_P$: The only recursive call is in Line 7 for subgoals $SG \in SGs(I)$. As $SG \prec_P I \prec_P G$, each recursive call terminates by induction hypothesis. Since all loops terminate, procedure *calc-contrib* is terminating. $\square$

Lemma 7.2 Let $\mathcal{IS}$ be an inference system defined with principal part and context as in Definition 5.4. Then, for each goal G $contribI(G) \prec_P G$.

Proof. This is proved with induction on G w.r.t. $\prec_P$: $contribI(G)$ is set in Lines 3, 11 and 14 of procedure *calc-contrib*(G) (cf. Figure 7.4), respectively.

- In Lines 3 and 14, $contribI(G)$ is set to $I = SI(G) \prec_P G$. Thus, $contribI(G) \prec_P G$ in these cases.

- In Line 11, $contribI(G)$ is set to $contribI(SG)$ for a subgoal $SG \in \mathcal{G} \subseteq SGs(I)$. Thus, $SG \prec_P I \prec_P G$. By induction hypothesis for SG, we get $contribI(G) = contribI(SG) \prec_P SG \prec_P G$.

$\square$

Lemma 7.3 Let $\mathcal{IS}$ be an inference system defined with principal part and context as in Definition 5.4 and $I = \mathit{contribI}(G)$. Then the following holds true:

(a) $\mathit{contrib}(PG(I)) = \mathit{contrib}(G) \subseteq G$;

(b) $\mathit{princ}(I) \subseteq \mathit{contrib}(G)$;

(c) for each $i \in \mathit{nbSGs}(I)$:
$\mathit{contrib}(SG_i(I)) \subseteq \mathit{side}(I,i) + \mathit{non\text{-}side}(I,i,\mathit{contrib}(PG(I)) - \mathit{princ}(I))$.

Proof. The following technical proof depends on Properties (5.2) to (5.11) defined for the operations on goals in Section 5.2.

We perform induction on G w.r.t. $\prec$: $\mathit{contrib}(G)$ is set in Lines 4, 12 and 15 of procedure $\mathit{calc\text{-}contrib}(G)$ (cf. Figure 7.4), respectively.

(a) We first prove $\mathit{contrib}(PG(I)) = \mathit{contrib}(G)$:

- In Lines 4 and 15, $PG(I) = G$ because $\mathit{contribI}(G)$ is set to $SI(G)$ in Lines 3 and 14, respectively. Thus, $\mathit{contrib}(PG(I)) = \mathit{contrib}(G)$ in these cases.
- In Line 12, $\mathit{contrib}(G)$ is set to $\mathit{contrib}(SG)$ for a subgoal $SG \in \mathcal{G} \subseteq SGs(I)$. Thus, $SG \prec_P I \prec_P G$. By induction hypothesis for SG, we get
$\mathit{contrib}(PG(\mathit{contribI}(SG))) = \mathit{contrib}(SG)$.
Due to Line 11, $PG(\mathit{contribI}(SG)) = PG(\mathit{contribI}(G)) = PG(I)$.
Thus, we get $\mathit{contrib}(PG(I)) = \mathit{contrib}(G)$.

It remains to prove $\mathit{contrib}(G) \subseteq G$:

- In Line 4, $\mathit{contrib}(G)$ is set to $\mathit{princ}(I) \subseteq PG(I) = G$. Thus, $\mathit{contrib}(G) \subseteq G$ in this case.
- In Line 12, $\mathit{contrib}(G)$ is set to $\mathit{contrib}(SG)$ for a subgoal SG with $\mathit{contrib}(SG) \subseteq G$. Thus, $\mathit{contrib}(G) \subseteq G$ in this case.
- In Line 15, $\mathit{contrib}(G)$ is set to
$\mathit{princ}(I) + \bigcup_{i=1}^{\mathit{nbSGs}(I)} \mathit{nside\text{-}src}(I,i,\mathit{contrib}(SG_i(I)) - \mathit{side}(I,i))$.
Because of $PG(I) = G$, it suffices to prove
$\mathit{princ}(I) + \bigcup_{i=1}^{\mathit{nbSGs}(I)} \mathit{nside\text{-}src}(I,i,\mathit{contrib}(SG_i(I)) - \mathit{side}(I,i)) \subseteq PG(I)$.
Due to Definition 5.4(a) and Properties (5.7) and (5.10), this can be further reduced to $\bigcup_{i=1}^{\mathit{nbSGs}(I)} \mathit{nside\text{-}src}(I,i,\mathit{contrib}(SG_i(I)) - \mathit{side}(I,i)) \subseteq \mathit{context}(I)$.
Due to Property (5.2), it suffices to prove for each $i \in \{1,\ldots,\mathit{nbSGs}(I)\}$:
$\mathit{nside\text{-}src}(I,i,\mathit{contrib}(SG_i(I)) - \mathit{side}(I,i)) \subseteq \mathit{context}(I)$.

  By induction hypothesis for $SG_i(I) \prec_P G$, $\mathit{contrib}(SG_i(I)) \subseteq SG_i(I)$. With Definition 5.4(b2), we get $\mathit{contrib}(SG_i(I)) \subseteq \mathit{side}(I,i) + \mathit{non\text{-}side}(I,i,\mathit{context}(I))$. Because of Property (5.9), $\mathit{contrib}(SG_i(I)) - \mathit{side}(I,i) \subseteq \mathit{non\text{-}side}(I,i,\mathit{context}(I))$. With Definition 5.4(b3), the claim follows.

(b)
- In Line 4, $\mathit{contrib}(G)$ is set to $\mathit{princ}(I)$. Thus, $\mathit{princ}(I) \subseteq \mathit{contrib}(G)$ in this case.

- In Lines 11 and 12, $contribI(G)$ is set to $contribI(SG)$ and $contrib(G)$ is set to $contrib(SG)$ for a subgoal SG. By induction hypothesis for $SG \prec_P G$, we get $princ(contribI(SG)) \subseteq contrib(SG)$. Therefore, $princ(I) \subseteq contrib(G)$ in this case.
- In Line 15, $contrib(G)$ is set to
$princ(I) + \bigcup_{i=1}^{nbSGs(I)} nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i))$.
Thus, $princ(I) \subseteq contrib(G)$.

(c)

- In Line 4, $nbSGs(I) = 0$, i.e., there are no subgoals. Thus, nothing has to be shown.
- In Line 12, the claim follows from the induction hypothesis for $SG \prec_P G$ since $contrib(G) = contrib(SG)$ and $contribI(G) = contribI(SG)$.
- Due to (a) and Property (5.8), $contrib(SG_i(I)) - side(I, i) \subseteq SG_i(I) - side(I, i)$. With Definition 5.4(b2) and Property (5.9), we get $contrib(SG_i(I)) - side(I, i) \subseteq non\text{-}side(I, i, context(I))$. Because of Definition 5.4(b3), we get

$$nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i)) \subseteq context(I) \tag{7.1}$$

$$\begin{aligned} &non\text{-}side(I, i, nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i))) \\ &\qquad = contrib(SG_i(I)) - side(I, i) \end{aligned} \tag{7.2}$$

In Line 15, $PG(I) = G$ and
$contrib(PG(I)) = princ(I) + \bigcup_{i=1}^{nbSGs(I)} nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i))$.
Because of Properties (5.3) and (5.7), we get for each $i \in \{1, \ldots, nbSGs(I)\}$:
$princ(I) + nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i)) \subseteq contrib(PG(I))$.
Because of (7.1) and Definition 5.4(a), we can apply Property (5.11). Together with Property (5.8), this results in
$nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i)) =$
$(princ(I) + nside\text{-}src(I, i, contrib(SG_i(I)) - side(I, i))) - princ(I) \subseteq$
$contrib(PG(I)) - princ(I)$.
Thus, we get
$contrib(SG_i(I)) - side(I, i) \subseteq non\text{-}side(I, i, contrib(PG(I)) - princ(I))$ from (7.2) using Definition 5.4(b1). The claim follows from Property (5.9).

□

Definition 7.4 (Essentially Contributing Proof Steps / Elements w.r.t. ***selectCG*****)**
Let $\mathcal{IS}$ be an inference system defined with principal part and context as in Definition 5.4. Let P be a proof for goal G. Let $contrib(G)$ and $contribI(G)$ be computed with procedure *calc-contrib*(G) and heuristics *selectCG* as presented in Figure 7.4. Then, the elements in $contrib(G)$ are called the *(essentially) contributing elements* of G for P w.r.t. *selectCG*. Elements in $(G - contrib(G))$ are called *non-contributing*. The inference rule $I = SI(G)$ applied to G is called *(essentially) contributing* for P w.r.t. *selectCG* if $contribI(G) = I$. Otherwise, I is called *non-contributing*. □

$I \leftarrow contribI(G)$;
$I' \leftarrow adaptI(I, G')$;
apply inference rule I' *to goal* G';
for $j = 1$ **to** $nbSGs(I')$ **do**
  $i \leftarrow selectAG(I, I', SG_j(I'))$;
  reuse-proof$(SG_i(I), SG_j(I'))$;

Figure 7.5: **Procedure** *reuse-proof*(G, G') with Precondition $contrib(G) \subseteq G'$

7.3 Reuse

For adaptable inference systems (cf. Definition 5.5), we can exploit Lemma 7.3 to reuse the proof of a goal G for G' if $contrib(G) \subseteq G'$. For this, procedure *reuse-proof* adapts inference rule $I = contribI(G)$ to G' (cf. Figure 7.5). For each new subgoal $SG_j(I')$, we find one subgoal $SG_i(I)$ such that the precondition $contrib(SG_i(I)) \subseteq SG_j(I')$ for procedure *reuse-proof* is fulfilled. Thus, we can apply it recursively to $SG_i(I)$ and $SG_j(I')$. The following lemma states the soundness of procedure *reuse-proof*.

Lemma 7.5 Let $\mathcal{IS}$ be an adaptable inference system as defined in Definition 5.5. Let G be a proved goal and $contrib(G)$ as well as $contribI(G)$ be computed with procedure *calc-contrib* (cf. Figure 7.4). If $contrib(G) \subseteq G'$ then the following properties hold true:

(a) each recursive call *reuse-proof*$(SG_i(I), SG_j(I'))$ obeys the precondition $contrib(SG_i(I)) \subseteq SG_j(I')$;

(b) the call of *reuse-proof*(G, G') is terminating;

(c) *reuse-proof*(G, G') creates a closed proof state tree for G'.

Proof.

(a) Due to Lemma 7.3(b), $princ(I) \subseteq contrib(G)$. From the assumption $contrib(G) \subseteq G'$, we get $princ(I) \subseteq G'$. According to Definition 5.5, function $adaptI(I, G')$ returns an inference rule I' that can be applied to G'. Furthermore, $selectAG(I, I', SG_j(I'))$ chooses for each subgoal $SG_j(I')$ an index i such that for the subgoal $SG_i(I)$ the following holds true: $side(I, i) + non\text{-}side(I, i, (PG(I) \cap G') - princ(I)) \subseteq SG_j(I')$.

We have to prove: $contrib(SG_i(I)) \subseteq SG_j(I')$. Because of Lemma 7.3(c), it suffices to prove $side(I, i) + non\text{-}side(I, i, contrib(PG(I)) - princ(I)) \subseteq SG_j(I')$.

Due to Lemma 7.3(a), $contrib(PG(I)) \subseteq PG(I)$. Because of Lemma 7.3(a) and the assumption, $contrib(PG(I)) = contrib(G) \subseteq G'$.
Therefore, $contrib(PG(I)) \subseteq PG(I) \cap G'$ because of Property (5.4).
Due to Property (5.8), $contrib(PG(I)) - princ(I) \subseteq (PG(I) \cap G') - princ(I)$. Thus, the claim follows from Definition 5.4(b1) and Property (5.7). Note that $(PG(I) \cap G') - princ(I) \subseteq PG(I) - princ(I) \subseteq context(I)$ because of Definition 5.4(a) and Properties (5.5), (5.9) and (5.10).

(b) This holds true since in each recursive call, the first argument decreases w.r.t. $\prec_P$, i.e. $SG_i(I) \prec_P G$.

(c) This holds true as G' and each of its subgoals is handled with recursive calls to procedure *reuse-proof* and this computation terminates.

□

Note that we may call function *reuse-proof* with the same goal G for both arguments. The call *reuse-proof*(G, G) replaces the former proof for G with its pruned proof where all non-contributing proof steps are eliminated.

7.3.1 Upward Propagation and Sideward Reuse

For the usefulness of our reuse mechanism it is crucial that we can efficiently find a proved goal G and an unproved goal G' such that $contrib(G) \subseteq G'$. We do not perform this search arbitrarily, but we fix one goal and perform this search in two directions: upwards and sidewards in the proof state tree. Our upward propagation fixes the proved goal G. It is very efficient because we try to propagate the (contributing) proof for G only to its unproved ancestor goals as far as possible. During sideward reuse, however, we fix one unproved goal G' and test for each proved goal G whose first proof step $SI(G)$ is contributing whether its (contributing) proof can be reused for G'. Note that we do not have to consider a goal G whose first proof step $SI(G)$ is non-contributing because the same proof is performed for $PG(contribI(G))$, i.e. for the father of the contributing inference rule $contribI(G)$ that can be adapted to G. To perform the sideward check efficiently, we use indexing techniques [RSV01]. In our current implementation, we use a simple trie-like data structure [Fre60]. The leaves of the trie are labeled with the proved goals. The path to goal G is labeled with the contributing elements $contrib(G)$. Since in a trie common prefixes are shared, we may exclude many goals at once if one of their contributing elements is not present in G'.

We do not present any further technical details for an efficient implementation of the checks for upward propagation and sideward reuse. Instead, we assume some abstract functions that perform these tasks:

- Upward propagation: Let *propagate-proof-p*(G) be a boolean valued function that returns **true** if the proof for goal G (or the proof for an ancestor goal of G whose proof is completed by proving G) can be propagated upwards in the proof state tree to an unproved ancestor goal G' of G. As a side-effect, if function *propagate-proof-p*(G) returns **true**, it stores G' in a global variable whose value is returned by function *propagated-goal*().

  Function *propagate-proof-p*(G) has to calculate the contribution of the newly proved goals. This information may be stored in a trie-like data structure to support sideward reuse (cf. Example 7.6).

- Sideward reuse: Let *reusable-proof-p*(G') be a boolean valued function that returns **true** if there is a proved goal G whose proof can be reused for G'. As a side-effect, if function *reusable-proof-p*(G') returns **true**, it stores G in a global variable whose value is returned by function *reusable-goal*().

7.3.2 Integration into a Simple Waterfall

In this section, we want to demonstrate that our approach can be integrated easily into a waterfall model for proof control. For this, we enhance the simple waterfall—modeled with function *simple-waterfall* in Section 3.2.1—with our new reuse mechanisms. Whereas we integrate upward propagation directly into the waterfall resulting in a new control structure *simple-waterfall-with-reuse* (cf. Figure 7.6), sideward reuse is modeled with a separate phase *reuse-phase* (cf. Figure 7.7).

First, we illustrate our integration with the abstract example presented in Section 7.1. Then, we comment on the pseudo code which sketches the integration.

Example 7.6 To simplify matters, we assume that the simple waterfall *simple-waterfall* (cf. Section 3.2.1) generates the proof state tree presented in Figure 7.1 in the following way:

- The inference rules are applied in ascending order.
- Each successful phase applies exactly one inference rule.
- All the subgoals generated by an inference rule are put into the pool of open goals and handled with recursive calls to the waterfall.

Thus, the first successful phase of *simple-waterfall* applies inference rule I_1 to goal G_1. The generated subgoals G_2 and G_{11} are handled with recursive calls of *simple-waterfall*. The proof state tree corresponds to the dynamic call structure of *simple-waterfall*. The handling of G_3 in *simple-waterfall*(G_3, phases), for instance, is initiated by *simple-waterfall*(G_2, phases) which, in turn, is called by *simple-waterfall*(G_1, phases).

The new waterfall *simple-waterfall-with-reuse* differs from *simple-waterfall* only after the first subgoal has been proved. As soon as a subgoal is proved, *simple-waterfall-with-reuse*

- computes the contribution of all newly proved goals.
- tries to propagate the proof of the highest newly proved goal upwards in the proof state tree.
- enables sideward reuse for all newly proved goals whose first proof step is contributing by storing the goals in a trie-like data structure. Sideward reuse itself is integrated into the waterfall as a separate phase *reuse-phase*. This is possible because we want to apply it only to open goals which are leaves of the proof state tree. In the waterfall presented in Section 3.2.2.4, it is sensible to apply *reuse-phase* before or just after `prove-taut` which tests for simple tautologies.

For the proof state tree presented in Figure 7.1, the following actions are performed:

- Until goal G_5 has been proved, nothing changes: Sideward reuse is checked during each call of the waterfall but it is not applicable because none of the goals is proved up to now.

```
foreach p ∈ phases do
    if apply-phase(p, G, 𝒢′) then
        if 𝒢′ = ∅ then
            if propagate-proof-p(G) then
                throw propagation;
            else
                return ∅;
        else
            𝒢 ← ∅;
            while 𝒢′ ≠ ∅ do
                choose a goal G′ ∈ 𝒢′;
                𝒢′ ← 𝒢′ − {G′};
                try
                    𝒢″ ← simple-waterfall-with-reuse(G′, phases);
                catch propagation
                    G_p ← propagated-goal();
                    if G ≺_P G_p then
                        throw propagation;
                    else if G = G_p then
                        reuse-proof(G_p, G_p);
                        return ∅;
                    else
                        𝒢 ← {G″ ∈ 𝒢 | G″ ⊀_P G_p};
                        𝒢′ ← {G″ ∈ 𝒢′ | G″ ⊀_P G_p};
                        𝒢″ ← ∅;
                        reuse-proof(G_p, G_p);
                𝒢 ← 𝒢 + 𝒢″;
            return 𝒢;
return {G};
```

Figure 7.6: **Function** *simple-waterfall-with-reuse*(G, *phases*)

```
if reusable-proof-p(G) then
    reuse-proof(reusable-goal(), G);
    𝒢 ← ∅;
    return true;
else
    return false;
```

Figure 7.7: **Function** *reuse-phase*(G, `REF` $\mathcal{G}$)

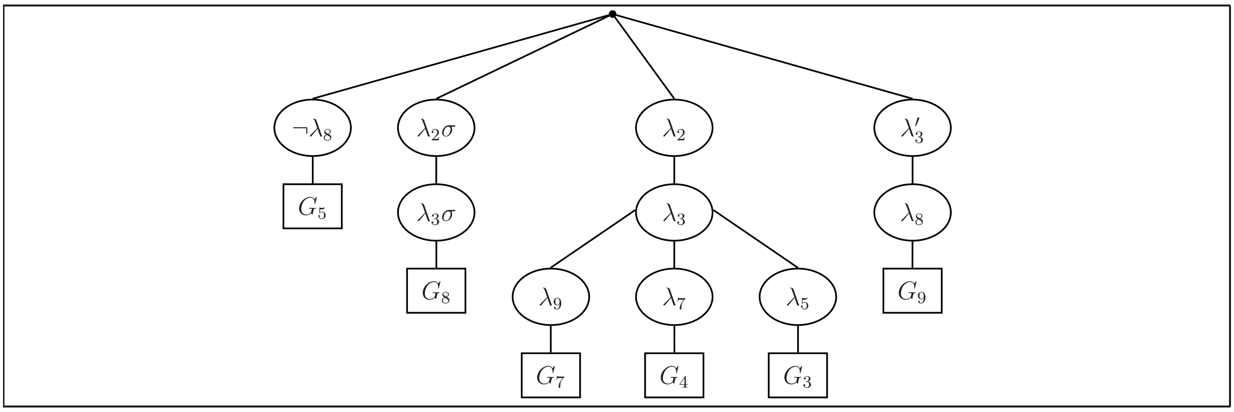

Figure 7.8: Trie-like Data Structure for Sideward Reuse

- As soon as G_5 is proved, the applicability of upward propagation is checked by calling *propagate-proof-p*(G_5). This function computes the contribution w.r.t. all goals that are newly proved which in this case applies only to G_5. Since $\neg\lambda_8$—the contributing element in G_5 (cf. Table 7.1 in Section 7.1)—is not present in the parent goal G_4, upward propagation is not applicable.

  To support sideward reuse, G_5 and its contributing elements are stored in a trie-like data structure. In Figure 7.8, we sketch one possible instance of this data structure after having inserted the parent goals of all contributing proof steps. Right now, only the left-most branch is inserted. It indicates that the proof of G_5 can be adapted to a goal G if only the contributing element $\neg\lambda_8$ is present in G. This property is checked by *reuse-phase* for all goals that are handled by the waterfall subsequently. But, since $\neg\lambda_8$ is not present in any other goal, the proof cannot be reused in this proof state tree.

- The next proof of a subgoal is completed by applying inference rule I_8. Beside G_8, this application also proves G_7 and G_6. The call of *propagate-proof-p*(G_8) computes the contribution of these three goals and tries to propagate the proof of the highest goal G_6 upwards in the proof state tree which is not possible in this case. The trie-like data structure used for sideward reuse is supplemented with G_8 and G_7 and their contributing elements (cf. Figure 7.8). Note that G_6 is not added because proof step I_6 performed for G_6 is not contributing.

- With the application of I_9, goals G_9, G_4 and G_3 are proved. This time, the contributing elements in G_3 are present in its parent goal G_2. Therefore, the proof can be propagated upwards in the proof state tree. The call of *propagate-proof-p*(G_9) returns **true**. Goal G_2 is stored in a global variable whose value is returned by function *propagated-goal*(). Furthermore, goals G_9, G_4 and G_3 are added to the trie-like data structure (cf. Figure 7.8).

  Right now, we are still handling goal G_9 but we know that we have found a proof that can be used for an ancestor goal of G_9. Therefore, we wish to abort all recursive calls of the waterfall up to the ancestor goal that can be proved. Due to this abortion, we have to integrate upward propagation directly into the control structure of the waterfall—instead of implementing it as a separate phase which would be much easier. With the abortion, we avoid to handle open goals that are eliminated in the pruned proof that results from upward propagation. For the implementation of upward propagation, we use an exception handling mechanism (see below).

- Goal G_2 is proved by upward propagation. For this, the pruned proof for G_3 is reused.
- After applying inference rule I_{11}, the last open goal is G_{12}. According to the trie-like data structure in Figure 7.8 sideward reuse is applicable, since the contributing elements in G_4—namely λ_2, λ_3 and λ_7—are present in G_{12}. Therefore, function *reusable-proof-p*(G_{12}) returns true. Goal G_4 is stored in a global variable which is returned by function *reusable-goal*(). This information is exploited by phase *reuse-phase* which adapts the proof for G_4 to G_{12} and closes the whole proof state tree.

□

Sideward reuse is realized as a separate phase *reuse-phase*$(G, \texttt{REF}\ \mathcal{G})$ (cf. Figure 7.7). With the introduced functions, its implementation is straightforward: By calling *reusable-proof-p*, it looks for a proved goal whose proof can be reused for G. In this case, G can be proved by reusing the proof for the goal returned by *reusable-goal*. Thus, the set of new subgoals is empty and the function returns **true**. Otherwise, it returns **false**.

The realization of upward propagation in the new waterfall modeled with function *simple-waterfall-with-reuse* is more difficult as a proof may be propagated for more than one level (cf. Figure 7.6). We have to take care that we remove those goals from the pool that are affected by upward propagation, and that we transfer control to the right place in the recursive call stack of function *simple-waterfall-with-reuse*. For ease of presentation, we assume an exception handling mechanism with **throw**, **try** and **catch** to transfer control. This mechanism can be found in many programming languages such as `C++` [Str00] or `Java` [GJSB05]. Similar mechanisms are also available in COMMON LISP using just the keywords **throw** and **catch** [Ste99] and in `ML` using keywords **raise** and **handle** [Pau96].

As in *simple-waterfall* (cf. Section 3.2.1), our new control function *simple-waterfall-with-reuse* considers each phase of the waterfall successively until one of them can be applied successfully to the input goal G. In this case, variable $\mathcal{G}'$ contains the new subgoals resulting from the successfully applied phase. These subgoals have to be handled by recursive calls of *simple-waterfall-with-reuse* (cf. Line 9–28). The resulting subgoals are collected in variable $\mathcal{G}$ and returned at the end of *simple-waterfall-with-reuse*. Otherwise, if none of the phases can be applied to G, a set consisting of G itself is returned.

Upward propagation is realized in function *simple-waterfall-with-reuse* as follows: If the application of a phase proves the input goal G—i.e. the set of new subgoals $\mathcal{G}'$ is empty—we try to propagate the proof upwards in the proof state tree (Line 3–7). The applicability of upward propagation is checked by calling function *propagate-proof-p*. If the proof can be propagated upwards, we throw an exception *propagation*. This exception is caught in each recursive call of *simple-waterfall-with-reuse* (Line 13–26). Let G_p be the goal to which the proof can be propagated. Since G_p can be proved, we do not have to care about any of its offsprings even if they represent open proof obligations. These are eliminated by pruning the proof state tree. Therefore, we consider the following cases:

- If G_p is an ancestor of goal G for which *simple-waterfall-with-reuse* is called, we pass the *propagation* exception on (Line 17–18). All open proof obligations in $\mathcal{G}$ and $\mathcal{G}'$ are automatically eliminated from the proof state tree when it is pruned by propagating the contributing proof upwards to the ancestor goal.

- If G_p is equal to G, the input goal is proved by upward propagation. Therefore, function *simple-waterfall-with-reuse* returns an empty set of new subgoals (Line 19–21).

- Otherwise, G_p is an offspring of G (Line 22–26). Therefore, G is not proved by upward propagation w.r.t. G_p. But still, there may be some offsprings of G_p in the pool given by $\mathcal{G}$ and $\mathcal{G}'$. As we do not have to consider these offsprings anymore they are eliminated in Line 23–24. We assume that at least goal G'—for which the waterfall has been called recursively—is proved by upward propagation. Otherwise, the exception would have been handled previously. In our realization, the actual upward propagation is applied only after all affected goals have been removed from the pool.

7.4 Case Studies

In this section, we validate our reuse mechanisms with some case studies. At the same time, we present the final version of our proof control for QuodLibet developed in this thesis. Due to Lemma 5.7, the inference system of QuodLibet is adaptable and, thus, our reuse mechanisms are applicable.

As a starting point for the validation we use Configuration (D) of Section 4.3.1. In this configuration, linear arithmetic is built-in; proof search is controlled with mandatory, obligatory and generous literals. We enhance the proof control with upward propagation and sideward reuse according to Section 7.3. For the integration, we have to answer one additional important question: Which of the proved goals are considered for sideward reuse? On the one hand, if we consider too many goals, the search for an appropriate one may be too time-consuming. On the other hand, if we consider too few goals, sideward reuse is rarely applicable. As a compromise, we consider exactly the proved goals in the same proof state tree for sideward reuse. This is justified because the goals in one proof state tree usually share some elements as they are derived from a common root goal. Therefore, we get a high probability that we may reuse the proof of a goal within one proof state tree. Between arbitrary proof state trees, this property is not guaranteed. Therefore, the applicability of sideward reuse is by far less probable.

With our implementation of the reuse mechanism, we aim at *reducing the runtime* for *creating* closed proof state trees as much as possible. On the one hand, we prune a proof state tree by propagating a proof upwards only if this eliminates some proof obligations. Therefore, we could derive much shorter final proofs if we applied further pruning. This is sensible if proofs are presented to human users or proof checkers. On the other hand, we do not only *mark* a goal as proved if we can apply the reuse mechanism to it but we really *carry out* the proof for the goal by adaptation. Thus, we may speed up the computation if we are interested only in whether a lemma is valid but not in a proof itself. Instead of "replaying" a proof each time it can be reused, we could also introduce new lemmas consisting of the contributing elements in the root goal of the reusable proof. This may

Str. Lemmas	Reuse	UP	SR	Autom. Appl.	Del.	Fin. P.	Runtime	
Example `sortalgos`								
0 / 111	—	—	—	2253	49	2204	6.66	+6.3%
	✓	1	120	2216	54	2162	7.08	
Example `gcd`								
1 / 57	—	—	—	830	10	820	3.85	-8.3%
	✓	0	35	819	10	809	3.53	
Example `exp-exhelp`								
0 / 18	—	—	—	675	0	675	2.70	-5.9%
	✓	0	53	659	0	659	2.54	
Example `sqrt` (H)								
0 / 18	—	—	—	352	23	329	1.51	-16.6%
	✓	0	18	297	23	274	1.26	
Example `sqrt` (E)								
0 / 34	—	—	—	464	8	456	1.44	+6.2%
	✓	1	19	466	13	453	1.53	
Example `Lpo` (A)								
23 / 273	—	—	—	20270	2583	17687	369.73	-32.2%
	✓	41	1993	18637	2715	15922	250.79	
Example `f91`								
1 / 11	—	—	—	443	13	430	1.85	-21.1%
	✓	0	36	392	0	392	1.46	
Example `mjrty`								
0 / 8	—	—	—	693	119	574	3.39	-43.4%
	✓	0	67	553	3	550	1.92	

Table 7.2: Statistics for Case Studies with and without Reuse using Heuristics **{m,o}**

speed up the computation if a proof is reused more than once. But as this would blow up the lemma data base, we have not investigated this variant, yet.

For the comparison we use the case studies presented in Sections 4.3.1 and 6.3. In the case study about the LPO, we use the full version (denoted by (A) for all) as presented in Chapter 8 and Appendix A. As an additional case study, we use

`mjrty`: In this example, we prove the soundness of a majority voting protocol according to [BM91].

Table 7.2 contains the statistics for the case studies with (resp. without) reuse enabled as indicated in column "Reuse" with "✓" (resp. "—"). For each case study, column "Str. Lemmas" contains two entries. The second entry is the number of lemmas proved, the first one the number of lemmas that can be *strengthened* by eliminating non-contributing literals. The remaining columns are split into two lines. The first (resp. second) line presents the statistics without (resp. with) reuse enabled. Column "UP" contains the number of applications of upward propagation, column "SR" the number of applications of sideward reuse. As for the statistics in Sections 4.3.1 and 6.3, the remaining columns contain the number of inference rules applied during the whole proof search, the number of inference rules deleted, the number of inference rules in the final proof, and the runtime in seconds measured by a CMU Common Lisp system on a machine with a 1 GHz Intel III processor and 4 GB RAM. Now, an inference rule may be deleted for two reasons: firstly, if the proof search gets stuck and has to backtrack; secondly, if a proof is propagated upwards.

From the statistics in Table 7.2, we draw the following conclusions:

- In the case studies, the final proofs become shorter when applying the reuse mechanism. As already mentioned, we apply upward propagation only if this eliminates some proof obligations. Therefore, we may get much shorter proofs if we apply upward propagation at the end of each proof once more.
- The reuse mechanism introduces some overhead. Thus, the runtime may increase with reuse enabled
  - if the heuristics for proof search are well suited for the case study: this applies to `sqrt` (E); or
  - if the case study consists of very simple lemmas only: this applies to `sortalgos`.

  But typically, with our reuse mechanism, the runtime decreases by 15 to 30% for our case studies. For some case studies, we even get speed-ups of more than 40%.
- Upward propagation is rarely used in the case studies. The main reason for this—beside the reason given in the first item—is that our proof search based on a mandatory marking favors proof steps that use new literals. Therefore, these proof steps are locally contributing and can rarely be eliminated. Thus, we expect greater benefits for systems that apply other heuristics for guiding proof search. We will comment on this claim below.

  Note, however, that upward propagation is frequently applicable in the case study about the LPO. This case study is by far the most complicated one. Furthermore, it is the only case study where we exploit a generous marking which admits proof steps that do not locally contribute. We discuss the interplay between generous markings and upward propagation in Section 8.2.2.3.
- Of 530 lemmas in total 25 lemmas can be strengthened by eliminating non-contributing literals. This is astonishing as we do not speculate lemmas automatically. But as the case studies indicate, a user may get lost without computer assistance when speculating lemmas for complicated examples manually. For lemmas speculated automatically, we expect an even higher degree of lemmas that can be strengthened.

We now comment on the claim stated in the third item, namely, that we expect greater benefits for systems that apply other heuristics for guiding proof search. For this, we apply our reuse techniques to the case studies in Section 6.3 without restricting proof search with markings, i.e. with heuristics $\varnothing$ in Table 6.2. Even in this case proof steps that involve new literals are preferred but other proof steps are not excluded from proof search as it is done with the strict mandatory markings heuristics. The results are summarized in Table 7.3.

Reuse	Open Lemmas	UP	SR	Autom. Appl.	Del.	Fin. P.	Runtime	
Example `sortalgos`								
—	2	—	—	2348	143	2205	6.77	+2.8%
✓	1	18	109	2351	287	2064	6.96	
Example `exp-exhelp`								
—	0	—	—	3290	9	3281	299.28	-22.2%
✓	0	33	193	2747	1448	1299	232.90	
Example `sqrt` (H)								
—	1	—	—	2008	969	1039	16.11	+6.3%
✓	1	3	87	2018	1005	1013	17.12	
Example `sqrt` (E)								
—	0	—	—	477	4	473	1.24	-1.6%
✓	0	0	23	477	4	473	1.22	
Example `Lpo`								
—	1	—	—	11125	1089	10036	291.37	-76.5%
✓	1	102	520	6952	2435	4517	68.38	

Table 7.3: Statistics for Case Studies with and without Reuse using Heuristics $\varnothing$

- For `sortalgos` and `sqrt` (E), neither between the different search heuristics (cf. Table 6.2) nor w.r.t. the usage of the reuse mechanisms applied to heuristics **{m,o}** (cf. Table 7.2), there is major difference w.r.t. the efficiency of proof search. Therefore, it is not astonishing that this situation does not change with heuristics $\varnothing$ (cf. Table 7.3). Note, however, that one additional lemma of `sortalgos` can be proved with reuse mechanisms.
- With regard to the other three case studies, only the efficiency of `sqrt` (H) is not improved with reuse enabled (cf. Table 7.3). For `exp-exhelp` and `Lpo`, the speed-up is substantial. Note the correlation between the achieved speed-up and the number of applications of upward propagation. Therefore, it seems to be beneficial to improve the applicability of upward propagation. This is subject of further research.

To summarize, with our reuse mechanisms we get simpler proofs, normally shorter runtimes and may strengthen some of the lemmas automatically.

7.5 Related Work

We have developed our reuse mechanisms independently from other approaches. On a very abstract level, however, the basic ideas have been successfully used in other research areas as e.g. `SAT` (*boolean satisfiability*) based on `DPLL`, `CSP` (*constraint satisfaction*) and `LP` (*logic programming*). In all these application domains, extensive search algorithms based on backtracking are employed. The basic idea is to improve the efficiency of search by learning from previous attempts. More precisely, previous attempts are analyzed, the essential information is extracted and reused. This is done

- to improve the backtracking mechanism: We call this upward propagation. It is called *non-chronological backtracking* for SAT, *conflict-directed backjumping* for CSP and *intelligent backtracking* for LP.
- to handle similar situations during the subsequent search: We call this sideward reuse. It is called *conflict-driven learning* for SAT, *no-goods* for CSP and *selective reset* for LP.

Although the basic ideas are the same on an abstract level, the approaches vastly differ in their specific implementation. This is caused by the different application domains and their intended usage. In the following, we point out some of these differences w.r.t. our approach.

The simplest of the problems mentioned above is SAT. It is concerned with deciding whether a given propositional formula is satisfiable. Provers based on the DPLL procedure [DLL62] systematically search the space of all variable assignments for a satisfying one. This search may be illustrated with a *semantic tree*. On each level, one variable is assigned a truth value. The problem arises in the exponential blow-up of the search space w.r.t. the number of different variables. *Non-chronological backtracking* and *conflict-driven learning* introduced in [MSS96] and [BS97] help to prune the search space. These techniques have dramatically increased the scope of SAT solvers based on DPLL. See [ZM02] for a survey.

A simple generalization of SAT is CSP. Instead of the two truth values, each variable may be assigned a value from a finite set. Nevertheless, the search space for CSP is still finite. *Conflict-driven backjumping* [Pro93] and the use of *no-goods* [SV93] laid the foundations for the approaches in SAT.

Both problems—SAT and CSP—have the same characteristics w.r.t. the search performed:

- A search step corresponds to an assignment of a variable. Therefore, their are *no* restrictions on performing these steps. The order in which variables are assigned a value may be chosen arbitrarily. This facilitates the implementation of the basic ideas.
- The search space is *finite*.
- *Negative* results are exploited for the reuse mechanisms: Backtracking is performed to a state which may be extended to a solution—i.e. a satisfying assignment. Previous conflicts may be stored to abort the search for those states that cannot be extended to a solution.

A more general problem is addressed in LP. In engines for logic programming languages such as PROLOG, a query is handled by resolution and unification of horn clauses over first-order logic. The nodes in the search tree are labeled with single literals. *Intelligent backtracking* aims at pruning the resulting search tree without losing any solutions. This technique can also be extended to non-horn clauses. See [Bru91] for a survey. The search may be characterized as follows:

- A search step corresponds to an application of a lemma. Therefore, search steps are restricted w.r.t. the applicability of the considered lemmas.

- In general, the search space is *infinite*.
- *Negative* results are exploited for the reuse mechanisms: If the unification problems resulting from lemma applications cannot be solved, backtracking is performed to a state which may be extended to a solution. As for `SAT` and `CSP`, previous conflicts may be stored to abort the search for those states that cannot be extended to a solution. This is modeled with a predicate at the meta-level and hyper-resolution in [Bru91].

There are even more application domains which exploit the basic ideas of reuse mentioned above. For tableaux, for instance, similar approaches have been proposed with the *factorization* and *folding up* rule. See [LS01] for a survey.

In this chapter, we have applied the basic ideas to adaptable inference systems—a special class of reductive inference systems working on proof state trees. In doing so, we do not restrict ourselves to concrete inference systems. Instead, we have presented a generic approach which is based on some easy requirements. The nodes in the proof search trees may consist of complex goals. In QuodLibet, a goal consists of two components—a clause and a weight. Our proof search has the following characteristics:

- A search step corresponds to an application of an inference rule. Therefore, search steps are restricted w.r.t. the applicability of the inference rules
- In general, the search space is *infinite*.
- *Positive* results are exploited for the reuse mechanisms: Instead of aborting proof attempts that cannot be completed, we try to complete proof attempts by reusing former proofs according to their contribution. Due to our generic approach and the complex structure of our goal nodes, the implementation of our reuse mechanisms is technically more involved. In particular, this holds true for computing the contribution of proofs which is required for the applicability of our reuse mechanisms.

To provide an efficient reuse mechanism, we have restricted ourselves to reusing pruned proofs without any modifications such as instantiation. Other reuse mechanisms such as the one described in [KW94] allow slight modifications of proofs. The combination of these approaches may be subject of further research. But it is questionable whether a reuse mechanism with higher complexity will pay off.

The application of our reuse mechanisms is not restricted to automatic proof attempts. Instead, it may also be applied for evaluating and improving proofs performed manually. The notion of contribution may be used for identifying the relevant proof steps in a proof. This is required for implementing intelligent tutoring systems.

Chapter 8

A Comprehensive Case Study: LPO

As explained in Chapter 1, different semantics exist for first-order logic. Whereas *deductive theorem proving* is concerned with the validity in all models, we are interested in *inductive theorem proving*. In contrast to deductive validity (in all models), inductive validity (e.g. in all data models) is not semi-decidable even for first-order logic. Thus, there is no hope for full automation and we have to cope with user-interaction.

Different inductive theorem provers vary in their interaction schemes. Furthermore, there is no agreement on the semantics of inductive theorem provers. Our semantics, for instance, is based on data models; other systems use the initial model of a specification. Therefore, different inductive theorem provers are difficult to compare. There does not exist a library of problems for inductive theorem provers such as the `TPTP` for deductive theorem provers [SS98]. Various attempts have been made towards this direction without success. One of the latest contains the following statement [Den05]:

> 'It is not practical for inductive theorem provers to follow the pattern of the `TPTP` library. Various attempts have been made to build a similar corpus of problems requiring inductive reasoning. The most mature of these was based on the Boyer-Moore corpus (This has become known as the Dmac corpus after David McAllester who translated a fragment of the `NQTHM` corpus into a simpler language.). This corpus was unpopular partly because there was repetition within the problem set and partly because many problems depended on a few particular function definitions. But the major objection was that inductive theorem provers use a number of different logics, some of which are typed and some of which are not, which made it difficult to agree on a standard format. The use of other logics also raised translation issues and a fully automated process for converting the theorems, even into an agreed typed language was never produced.'

As a consequence, it is problematic to compare different proof systems for inductive theorem proving. Furthermore, it is much more time-consuming to perform case studies for inductive theorem proving than for deductive theorem proving due to the required user-interaction. Therefore, we have performed only a few case studies within this thesis. Instead, we have focused on the development of new proof techniques which support user-interaction required for inductive theorem proving (cf. Chapters 4–7).

Our new proof techniques are particularly suited for complicated induction schemes as required for proofs about mutually recursive functions. Thus, we have concentrated on case studies that are specified with mutual recursion in a natural way. Note that specifications with mutual recursion pose problems to many other inductive theorem provers such as `NQTHM` [BM88a]. In these systems, mutually recursive function definitions are combined into one complex function definition. Then, the former functions are accessed with projections. But this results in complex proofs that are not human-oriented and in difficulties in supporting manual interactions.

In this chapter, we give a survey of our most challenging case study—a proof of the equivalence of various implementations of the lexicographic path order LPO [KL80] based on [Löc04]. This case study helped us to

- validate and refine some of our proof techniques,
- develop new proof techniques, and
- envision perspectives of further research which could not be worked out in this thesis.

Nevertheless, all the proof techniques proposed in this thesis are independent from a special application domain. They do not use any special knowledge, for instance, about the LPO. Instead, they are defined on a very abstract level such as the proof search heuristics based on markings presented in Chapter 6.

In Section 8.1, we give an overview on the application domain LPO. From this, we sketch the derivation of a proof script for QuodLibet in Section 8.2 focusing on the problems encountered in the case study. A *proof script* contains the complete input for QuodLibet consisting of the specifications, manual applications and calls to the automatic proof control. The output is called *proof log*. It contains information about all performed and deleted proof steps. The whole proof script for this case study can be found in Appendix A. In Section 8.3, we conclude this chapter pointing out some directions for further research.

8.1 The Application Domain

The case study about the LPO was initiated by Löchner [Löc04]. The LPO is in widespread use in automated theorem provers based on rewriting techniques such as Waldmeister [LH02]. In Waldmeister, up to 50% of the total running time is spent on order comparisons. Therefore, it is essential to provide an efficient implementation for these comparisons. From a standard definition of the LPO, Löchner derived an efficient implementation using program transformation techniques mainly based on some standard Unfold/Fold-calculus [BD77, PP93]. Whereas a straightforward implementation of the definition has exponential complexity his efficient implementation requires polynomial runtime only. To exclude bugs resulting from oversights during the transformation he wanted to prove the equivalence of the different implementations with a formal proof system such as QuodLibet. QuodLibet is particularly suited for this task as it allows for the specification of *partially defined* functions based on *mutual recursion* as well as the verification of properties about these functions in a natural way. Our case study results from a collaboration with Löchner. He accounted for the specifications of the different implementations as well as an initial proof plan in terms of proofs performed manually. From this, the author of this thesis derived a proof script for QuodLibet by adding auxiliary lemmas and hints e.g. for choosing suitable induction orders. Furthermore, the proof control of QuodLibet and its interaction scheme were improved. At the end, the case study was performed successfully, the manual interactions were reduced, and the runtime was reduced from more than 30000 seconds to nearly 250 seconds. We achieved these improvements without any "tricks". More details on the improvements can be found at the end of Section 8.2.1.3. With the formal proofs implicit assumptions in the informal proofs were identified. The reuse mechanisms described in Section 7 helped us to strengthen lemmas by eliminating unnecessary conditions.

As already mentioned in Section 2.2.3, the LPO provides a scheme of wellfounded orders on wellformed terms over function symbols with fixed arity and variable symbols. It depends on a *precedence* on function symbols. The following definition corresponds to the original one given in [KL80] which may also be found in [Ave95, BN98, Der87].

Definition 8.1 (The Lexicographic Path Order LPO) Let $>_F$ be a partial order on F and $t, u \in \mathrm{Term}(F, V)$. Then $t \succ_{\mathtt{lpo}} u$ iff either $t \equiv f(t_1, \ldots, t_n)$, $u \equiv g(u_1, \ldots, u_m)$, and

(α) $t_i \succeq_{\mathtt{lpo}} u$ for some $i \in \{1, \ldots, n\}$ or

(β) $f >_F g$ and $t \succ_{\mathtt{lpo}} u_k$ for each $k \in \{1, \ldots, m\}$ or

(γ) $f = g$, there is an $i \in \{1, \ldots, m\}$ such that $t_j \equiv u_j$ for each $j \in \{1, \ldots, i-1\}$ and $t_i \succ_{\mathtt{lpo}} u_i$, and $t \succ_{\mathtt{lpo}} u_k$ for each $k \in \{1, \ldots, m\}$

or $t \equiv f(t_1, \ldots, t_n)$, $u \in V$ and

(δ) $u \in V(t)$,

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

The derivation of an efficient implementation makes use of the fact that each LPO defines a simplification order [Der87].

Definition 8.2 (Simplification Orders) An order $\succ$ on terms is a *simplification order* if

- it is *monotonic*, i.e. if the sorts of t/p, u, v agree, $u \succ v$ implies $t[u]_p \succ t[v]_p$ for each $p \in Pos(t)$;
- it contains the *subterm order* $\succ_{\mathrm{ST}}$ where $u \succ_{\mathrm{ST}} v$ iff there exists a position $p \in Pos(u)$ with $p \not\equiv \varepsilon$ and $u/p \equiv v$.
- it fulfills the *deletion property*, i.e. $f(t_1, \ldots, t_n) \succ f(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$ for each $i \in \{1, \ldots, n\}$.[1]

□

[1] We consider terms over function symbols with fixed arity only. Therefore, the deletion property is not relevant for our case study about the LPO.

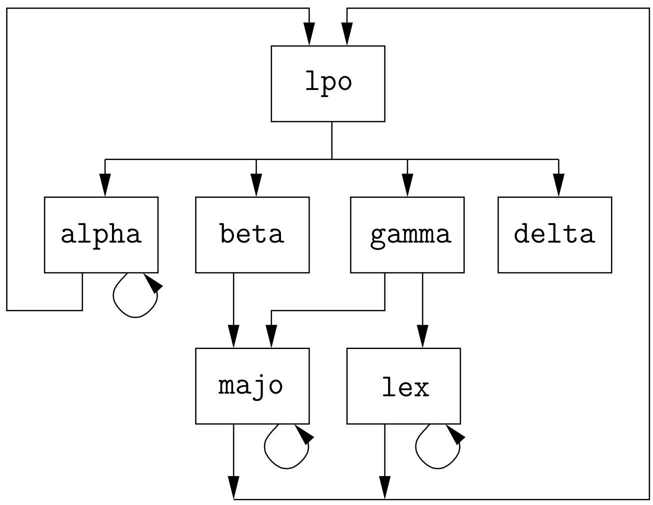

Figure 8.1: Recursive Dependencies in the Definition of `lpo`

Lemma 8.3 For each precedence $>_F$ over a set F of function symbols with fixed arity, the lexicographic path order $\succ_{\texttt{lpo}}$ is a simplification order. □

We have formally proved this lemma with QuodLibet within our case study (cf. Example 8.5 for more details on the proof of the transitivity and Section A.7 for the resulting proof script).

Definition 8.1 can be transformed into a functional program in a straightforward manner as it is done for variant $\texttt{lpo}_1$ below. The order comparisons are represented with boolean valued functions. The definition is realized with seven operators: An operator `lpo` which returns the final result of the comparison; operators `alpha`, `beta`, `gamma`, and `delta` which implement the four cases in the definition; and operators `majo` and `lex` which are used for the quantifications in cases (β) and (γ). This results in *recursive dependencies* between the operators as illustrated in Figure 8.1. Operators `alpha`, `beta`, `gamma`, and `delta`, for instance, are used in the definition of `lpo`; and `alpha` recursively calls `lpo` as well as itself.

During the derivation of an efficient implementation, six different variants of the LPO are defined. We distinguish the different variants by adding indices to the operators. Operator `alpha` in variant $\texttt{lpo}_2$, for instance, is denoted by $\texttt{alpha}_2$. An operator is defined in terms of operators of the *same* variant only. Often, two consecutive variants essentially differ in the definitions of a few operators only. The other definitions remain "the same", i.e. the definition of the latter variant is derived from the former one by replacing the operators of the former variant with the corresponding operators of the latter one. We do not present these implicit definitions. Therefore, we define only the first variant completely. For the other variants, we concentrate on those definitions that really differ from the previous variant. In a nutshell, the following variants are developed in [Löc04]:

$\texttt{lpo}_1$**:** This variant acts as reference implementation. It is derived from Definition 8.1 in a straightforward manner and has exponential runtime. In the functional program depicted in Figure 8.2, with $=_{\texttt{t}}$ (resp. $=_F$) two terms (resp. functions symbols) are compared w.r.t. syntactical equality. The boolean valued function $\texttt{contains}_{\texttt{tl}}$ returns true iff the variable in the second argument is contained in the first argument which is

$$\begin{aligned}
\mathtt{lpo}_1(\mathtt{F}(f,ts),\mathtt{F}(g,us)) &= \mathtt{alpha}_1(ts,\mathtt{F}(g,us)) \vee \mathtt{beta}_1(\mathtt{F}(f,ts),\mathtt{F}(g,us)) && (8.1)\\
&\qquad \vee\, \mathtt{gamma}_1(\mathtt{F}(f,ts),\mathtt{F}(g,us))\\
\mathtt{lpo}_1(\mathtt{F}(f,ts),\mathtt{V}(y)) &= \mathtt{delta}_1(\mathtt{F}(f,ts),\mathtt{V}(y)) && (8.2)\\
\mathtt{lpo}_1(\mathtt{V}(x),u) &= \mathtt{false} && (8.3)\\
\mathtt{alpha}_1(\mathtt{nil},u) &= \mathtt{false} && (8.4)\\
\mathtt{alpha}_1(\mathtt{cons}(t,ts),u) &= t =_{\mathtt{t}} u \vee \mathtt{lpo}_1(t,u) \vee \mathtt{alpha}_1(ts,u) && (8.5)\\
\mathtt{beta}_1(\mathtt{F}(f,ts),\mathtt{F}(g,us)) &= f >_F g \wedge \mathtt{majo}_1(\mathtt{F}(f,ts),us) && (8.6)\\
\mathtt{gamma}_1(\mathtt{F}(f,ts),\mathtt{F}(g,us)) &= f =_F g \wedge \mathtt{lex}_1(ts,us) \wedge \mathtt{majo}_1(\mathtt{F}(f,ts),us) && (8.7)\\
\mathtt{delta}_1(\mathtt{F}(f,ts),\mathtt{V}(y)) &= \mathtt{contains}_{\mathtt{tl}}(ts,y) && (8.8)\\
\mathtt{majo}_1(t,\mathtt{nil}) &= \mathtt{true} && (8.9)\\
\mathtt{majo}_1(t,\mathtt{cons}(u,us)) &= \mathtt{lpo}_1(t,u) \wedge \mathtt{majo}_1(t,us) && (8.10)\\
\mathtt{lex}_1(\mathtt{nil},\mathtt{nil}) &= \mathtt{false} && (8.11)\\
\mathtt{lex}_1(\mathtt{cons}(t,ts),\mathtt{cons}(u,us)) &= \mathbf{if}\ t =_{\mathtt{t}} u\ \mathbf{then}\ \mathtt{lex}_1(ts,us)\ \mathbf{else}\ \mathtt{lpo}_1(t,u) && (8.12)
\end{aligned}$$

Figure 8.2: Functional Program for Variant $\mathtt{lpo}_1$

a list of terms. It is defined by mutual recursion using another boolean valued function `contains` which performs the same check for a term as first argument. Using $=_V$ to compare two variables w.r.t. syntactical equality, this may be defined as follows:

$$\begin{aligned}
\mathtt{contains}(\mathtt{V}(x),y) &= x =_V y && (8.13)\\
\mathtt{contains}(\mathtt{F}(f,ts),y) &= \mathtt{contains}_{\mathtt{tl}}(ts,y) && (8.14)\\
\mathtt{contains}_{\mathtt{tl}}(\mathtt{nil},y) &= \mathtt{false} && (8.15)\\
\mathtt{contains}_{\mathtt{tl}}(\mathtt{cons}(t,ts),y) &= \mathtt{contains}(t,y) \vee \mathtt{contains}_{\mathtt{tl}}(ts,y) && (8.16)
\end{aligned}$$

Note that $\mathtt{lex}_1$ is a partial function. It is defined only for wellformed terms (cf. Example 8.4).

$\mathtt{lpo}_2$: This variant is derived from $\mathtt{lpo}_1$ by rearranging the order in which cases (α) to (γ) are checked. Therefore, only the definition of $\mathtt{lpo}_2$ changes in comparison to variant $\mathtt{lpo}_1$:

$$\begin{aligned}
\mathtt{lpo}_2(\mathtt{F}(f,ts),\mathtt{F}(g,us)) &= \mathtt{beta}_2(\mathtt{F}(f,ts),\mathtt{F}(g,us)) \vee \mathtt{gamma}_2(\mathtt{F}(f,ts),\mathtt{F}(g,us)) && (8.17)\\
&\qquad \vee\, \mathtt{alpha}_2(ts,\mathtt{F}(g,us))\\
\mathtt{lpo}_2(\mathtt{F}(f,ts),\mathtt{V}(y)) &= \mathtt{delta}_2(\mathtt{F}(f,ts),\mathtt{V}(y)) && (8.18)\\
\mathtt{lpo}_2(\mathtt{V}(x),u) &= \mathtt{false} && (8.19)
\end{aligned}$$

$\mathtt{lpo}_3$: In comparison to $\mathtt{lpo}_2$ this variant takes advantage of positive knowledge obtained from previous order comparisons for optimizing conjunctions of conditions. Therefore, it prevents some order comparisons by exploiting properties of simplification orders such as transitivity and containedness of the subterm relation.

More precisely, a new operator `lexM` is introduced which combines operators `lex` and `majo`. It is used in the definition of operator `gamma`:

$$\mathtt{gamma}_3(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = f =_F g \wedge \mathtt{lexM}_3(\mathtt{F}(f,ts),ts,us) \quad (8.20)$$

$$\mathtt{lexM}_3(t,\mathtt{nil},\mathtt{nil}) = \mathtt{false} \quad (8.21)$$

$$\begin{aligned}\mathtt{lexM}_3(t,\mathtt{cons}(t_i,ts),\mathtt{cons}(u_i,us)) = {} & \mathbf{if}\ t_i =_{\mathtt{t}} u_i\ \mathbf{then}\ \mathtt{lexM}_3(t,ts,us) \\ & \mathbf{else}\ \mathtt{lpo}_3(t_i,u_i) \wedge \mathtt{majo}_3(t,us)\end{aligned} \quad (8.22)$$

In variant $\mathtt{lpo}_3$, operator `majo` is applied only to subterms not already known to be smaller than t.

$\mathtt{lpo}_4$**:** This variant, additionally, makes use of knowledge about negative results from previous comparisons. These are used for optimizing disjunctions of conditions. For this optimization, recursive calls are unfolded and rearranged exploiting the properties of simplification orders once again, before being folded back. As shown in [Löc04], this variant has polynomial runtime because of the reduced number of `lpo` invocations via `alpha`.

In this variant, operators `alpha` and `lexM` are combined to a new operator `lexMA` which is used in the definition of $\mathtt{lpo}_4$:

$$\begin{aligned}\mathtt{lpo}_4(\mathtt{F}(f,ts),\mathtt{F}(g,us)) = {} & \mathbf{if}\ f >_F g\ \mathbf{then}\ \mathtt{majo}_4(\mathtt{F}(f,ts),us) \\ & \mathbf{elsif}\ f =_F g\ \mathbf{then}\ \mathtt{lexMA}_4(\mathtt{F}(f,ts),\mathtt{F}(g,us),ts,us) \\ & \mathbf{else}\ \mathtt{alpha}_4(ts,\mathtt{F}(g,us))\end{aligned} \quad (8.23)$$

$$\mathtt{lpo}_4(\mathtt{F}(f,ts),\mathtt{V}(y)) = \mathtt{delta}_4(\mathtt{F}(f,ts),\mathtt{V}(y)) \quad (8.24)$$

$$\mathtt{lpo}_4(\mathtt{V}(x),u) = \mathtt{false} \quad (8.25)$$

$$\mathtt{lexMA}_4(t,u,\mathtt{nil},\mathtt{nil}) = \mathtt{false} \quad (8.26)$$

$$\begin{aligned}\mathtt{lexMA}_4(t,u,\mathtt{cons}(t_i,ts),\mathtt{cons}(u_i,us)) = {} & \mathbf{if}\ t_i =_{\mathtt{t}} u_i\ \mathbf{then}\ \mathtt{lexMA}_4(t,u,ts,us) \\ & \mathbf{elsif}\ \mathtt{lpo}_4(t_i,u_i)\ \mathbf{then}\ \mathtt{majo}_4(t,us) \\ & \mathbf{else}\ \mathtt{alpha}_4(ts,u)\end{aligned} \quad (8.27)$$

$\mathtt{lpoR}_5$**:** For operators `alpha` and `lex`, we need information whether two terms are syntactically equal. Instead of two independent boolean valued checks for syntactic equality and the LPO, we may define one combined check called `lpoR` returning a three valued result of sort `Res`. The call $\mathtt{lpoR}(t,u)$ returns

- `E` if $t \equiv u$;
- `G` if $t \succ_{\mathtt{lpo}} u$; and
- `N` otherwise.

Variant $\mathtt{lpoR}_5$ optimizes this check by unfolding, rearranging, and folding back again. It is defined using new operators `lexMAE`, `majoR`, and `alphaR`. Operator `lexMAE` combines operator `lexMA` with a check for syntactic equality. Operators `majoR` and `alphaR` convert the interface of `majo` and `alpha` to sort `Res`. We skip the lengthy formal definition of these operators which can be found in [Löc04]. The corresponding specification for QuodLibet is presented in Section A.13.

$\texttt{clpo}_6$: In an application domain it is sometimes necessary to determine whether $t \succ_{\texttt{lpo}} u$ or $u \succ_{\texttt{lpo}} t$ holds true. We may combine these two comparisons to one bidirectional comparison. This is realized with function `clpo`. For this variant, sort `Res` is extended with `L`. The call $\texttt{clpo}(t, u)$ returns

- `E` if $t \equiv u$;
- `G` if $t \succ_{\texttt{lpo}} u$;
- `L` if $u \succ_{\texttt{lpo}} t$; and
- `N` otherwise.

Once again, unfold/fold techniques are used for deriving an optimized variant $\texttt{clpo}_6$. It depends on the new operators `cMA`, `cLMA`, and `cAA`. Roughly speaking, operator `cMA` performs a bidirectional comparison which combines operators `majo` and `alpha`. During the comparison of two terms t and u, it is called if the top-level operators of t and u are comparable and one of them is smaller than the other w.r.t. the considered precedence. Operator `cLMA` is the bidirectional version of operator `lexMA`. It performs the comparison if the top-level operators of t and u are equal. Operator `cAA` is the bidirectional version of operator `alpha`. It is called if the top-level operators of t and u are not comparable w.r.t. the considered precedence. Details on the formalization can be found in [Löc04] and Section A.17.

8.2 A Proof Script for QuodLibet

In informal proofs, properties such as the wellformedness of the arguments or the transitivity of simplification orders are often used implicitly. In formal proofs, these implicit proof steps have to be made explicit. This improves the accuracy of the proofs and possibly uncovers bugs, caused e.g. by missing assumptions.

In inductive theorem proving, some kind of interaction is usually required to perform proofs successfully. These interactions may be given in terms of auxiliary lemmas or more specific hints such as the provision of a suitable induction scheme or the manual application of a required lemma. For a successful proof attempt, the user usually has to think about a manual proof first. In a second step, the proof is transformed into the representation of the theorem prover. Imprecise informal arguments have to be worked out more precisely. The analysis of failed proof attempts helps in this endeavor.

Certainly, these two steps need not be separated in such a strict manner but their executions may be intertwined. Note that there are often many ways to formalize a problem. One formalization may be more suited for human understanding, another for implementation purposes, and yet another for performing formal proofs. The same holds true for informal versus formal proofs performed within a special theorem prover. The automatic proof control may guide proof search in another direction than that taken in the informal proof. Therefore, the initial manual proof plan may require some kind of revision. The intertwining of these two steps helps the user to learn from (un)successful previous proof attempts in such a way that suitable formalization and proof ideas can be found.

The realization of the case study about the LPO was quite challenging. Firstly, we had to find suitable auxiliary lemmas. Secondly, we had to implement new proof techniques in terms of new tactics and, actually, to improve the base system for efficiency reasons. Finally, we had to apply some tricks to complete the case study despite some deficiencies of the induction order. The integrated decision procedure for linear arithmetic helped us in this endeavor.

Because of the sheer size of this effort, we cannot present all the details of the proof engineering process. Nevertheless, we sketch this process in Section 8.2.1. In Section 8.2.2, we present details about the resulting proof script pointing out some problems and particularities of the case study as well as our solutions. The whole proof script is given in Appendix A.

8.2.1 The Proof Engineering Process

For the case study about the LPO, Löchner provided us with a preliminary version of his paper [Löc04] containing data structures, functional definitions of the defined operators as well as specifications of some auxiliary lemmas required for his manually performed proof sketches. This information could be used as proof plan. Note that the cited paper has been developed in parallel with the formal proofs. Therefore, the development of the paper and the derivation of the formal proofs discussed here influenced each other. As a consequence, the specifications in the final version of the paper can be used—via some straightforward translation—as the input specification for QuodLibet.

In the following sections, we describe the process that allowed us to produce the final proof script. We concentrate on those aspects that are specific for the derivation of formal proofs.

8.2.1.1 Data Structures

Most of the data structures required for this case study have already been introduced and motivated in Section 2.2.3: For the definition of terms, we require a sort `Term` for the representation of terms themselves, a sort `Termlist` for lists of terms used as arguments, and sorts `VID` and `FID` for variable and function symbols, respectively. In our case study, we consider function symbols with fixed arity and a total precedence only. This can be modeled using a constructor `Fid` for function symbols with two natural numbers as arguments: The first one is used for calculating the precedence, the second one contains the arity of the function symbol. As usually, we use sort `Bool` for boolean values and sort `Nat` for natural numbers. Furthermore, we define the following sorts:

`Position` for positions with constructors

- `nnil` : $\rightarrow$ `Position`
- `ncons` : `Nat`, `Position` $\rightarrow$ `Position`

Positions are represented as lists over natural numbers. They are used e.g. for the specification and verification that every LPO is a simplification order.

`Res` for the representation of the result for operators `lpoR5` and `clpo6` with constructors

- `E` : $\rightarrow$ `Res`

- $\mathtt{G} : \to \mathtt{Res}$
- $\mathtt{L} : \to \mathtt{Res}$
- $\mathtt{N} : \to \mathtt{Res}$

For the meaning of the constructors, we refer to Section 8.1.

8.2.1.2 Defining Rules

The functional programs presented in [Löc04] for the different variants of the LPO can be used as specifications in QuodLibet with just slight modifications. We illustrate this transformation with variant $\mathtt{lpo}_1$ illustrated in Figure 8.2 (cf. Section 8.1).

- For the equality checks $=_{\mathtt{t}}$ and $=_F$, we do not introduce new function symbols but use the predefined equality predicate. In doing so, we follow the paradigm that predefined symbols should be preferred in comparison to user-defined symbols as far as possible because, otherwise, humans would have problems (cf. [Wir05a]). Furthermore, the automatic proof control can make much more use of predefined symbols since they are fixed. A more detailed justification as well as a collection of further guidelines for developing suitable proof scripts for QuodLibet can be found in [SS04].

  Note that the representation of variable and function symbols is unique. Therefore, it is possible to use the predefined equality predicate for the comparison of terms.

- For if-then-else statements, we use the following transformation:

  $$l = \textbf{if } \lambda \textbf{ then } r_1 \textbf{ else } r_2 \quad \longrightarrow \quad \begin{array}{l} \{\ l = r_1,\ \ \neg\lambda\ \} \\ \{\ l = r_2,\ \ \lambda\ \} \end{array}$$

  This means that an if-then-else statement produces two defining rules which differ according to condition λ.

- For the boolean connectives $\vee$ and $\wedge$, the precedence $>_F$, and functions `contains` and $\mathtt{contains}_{\mathtt{tl}}$, we provide new function symbols `or`, `and`, `prec`, `contains` and `contains_tl`, respectively.

  We could also use the specification of $\vee$ and $\wedge$ in [Löc04], namely

  $$b1 \vee b2 = \textbf{if } b1 \textbf{ then } \mathtt{true} \textbf{ else } b2 \tag{8.28}$$
  $$b1 \wedge b2 = \textbf{if } b1 \textbf{ then } b2 \textbf{ else } \mathtt{false} \tag{8.29}$$

  and then transform the if-then-else statement as above. This would allow us to replace the user-defined symbols `or` and `and` in favor of some more defining rules which exploit the predefined disjunction of the clause form. In fact, we use both formalizations:

  – With `lpo1`, we try to mimic the original specification for $\mathtt{lpo}_1$ as close as possible for two reasons: Firstly, we try to avoid any unnecessary modifications of the specifications in [Löc04] even if they are quite obvious. Instead, we want to prove the properties for the original specification as far as possible. Secondly, we want to exploit properties of `and` and `or` such as associativity and commutativity.

Note that we have to perform the transformation in those cases where we use the predefined equality predicate instead of $=_{\mathtt{t}}$ and $=_F$. This has to be done since equality atoms cannot be used as arguments of function symbols.

– We also specify an internal representation Lpo which applies the transformations much more to support the automatic proof control. We prove that the LPO is a simplification order only for this internal representation. For the other variants this property follows from the equivalence proofs between the different variants and the internal representation.

Altogether, the transformation results in the following specification for lpo1:

$$\{\ \mathtt{lpo1}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) = \mathtt{or}(\mathtt{alpha1}(ts, \mathtt{F}(g, us)), \mathtt{or}(\mathtt{beta1}(\mathtt{F}(f, ts), \mathtt{F}(g, us)), \mathtt{gamma1}(\mathtt{F}(f, ts), \mathtt{F}(g, us))))\ \} \tag{8.30}$$

$$\{\ \mathtt{lpo1}(\mathtt{F}(f, ts), \mathtt{V}(y)) = \mathtt{delta1}(\mathtt{F}(f, ts), \mathtt{V}(y))\ \} \tag{8.31}$$

$$\{\ \mathtt{lpo1}(\mathtt{V}(x), u) = \mathtt{false}\ \} \tag{8.32}$$

$$\{\ \mathtt{alpha1}(\mathtt{nil}, u) = \mathtt{false}\ \} \tag{8.33}$$

$$\{\ \mathtt{alpha1}(\mathtt{cons}(t, ts), u) = \mathtt{true},\ t \neq u\ \} \tag{8.34}$$

$$\{\ \mathtt{alpha1}(\mathtt{cons}(t, ts), u) = \mathtt{or}(\mathtt{lpo1}(t, u), \mathtt{alpha1}(ts, u)),\ t = u\ \} \tag{8.35}$$

$$\{\ \mathtt{beta1}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) = \mathtt{and}(\mathtt{prec}(f, g), \mathtt{majo1}(\mathtt{F}(f, ts), us))\ \} \tag{8.36}$$

$$\{\ \mathtt{gamma1}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) = \mathtt{and}(\mathtt{lex1}(ts, us), \mathtt{majo1}(\mathtt{F}(f, ts), us)),\ f \neq g\ \} \tag{8.37}$$

$$\{\ \mathtt{gamma1}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) = \mathtt{false},\ f = g\ \} \tag{8.38}$$

$$\{\ \mathtt{delta1}(\mathtt{F}(f, ts), \mathtt{V}(y)) = \mathtt{contains\_tl}(ts, y)\ \} \tag{8.39}$$

$$\{\ \mathtt{majo1}(t, \mathtt{nil}) = \mathtt{true}\ \} \tag{8.40}$$

$$\{\ \mathtt{majo1}(t, \mathtt{cons}(u, us)) = \mathtt{and}(\mathtt{lpo1}(t, u), \mathtt{majo1}(t, us))\ \} \tag{8.41}$$

$$\{\ \mathtt{lex1}(\mathtt{nil}, \mathtt{nil}) = \mathtt{false}\ \} \tag{8.42}$$

$$\{\ \mathtt{lex1}(\mathtt{cons}(t, ts), \mathtt{cons}(u, us)) = \mathtt{lex1}(ts, us),\ t \neq u\ \} \tag{8.43}$$

$$\{\ \mathtt{lex1}(\mathtt{cons}(t, ts), \mathtt{cons}(u, us)) = \mathtt{lpo1}(t, u),\ t = u\ \} \tag{8.44}$$

and the following internal representation Lpo:

$$\{\ \mathtt{Lpo}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) = \mathtt{true},\ \mathtt{Alpha}(ts, \mathtt{F}(g, us)) \neq \mathtt{true}\ \} \tag{8.45}$$

$$\{\ \mathtt{Lpo}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) = \mathtt{true},\ \mathtt{Beta}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \neq \mathtt{true}\ \} \tag{8.46}$$

$$\{\ \texttt{Lpo}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{Gamma}(\texttt{F}(f,ts),\texttt{F}(g,us)),\ \texttt{Alpha}(ts,\texttt{F}(g,us)) = \texttt{true},\ \neg\texttt{def Alpha}(ts,\texttt{F}(g,us)),\ \texttt{Beta}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{true},\ \neg\texttt{def Beta}(\texttt{F}(f,ts),\texttt{F}(g,us))\ \} \tag{8.47}$$

$$\{\ \texttt{Lpo}(\texttt{F}(f,ts),\texttt{V}(y)) = \texttt{Delta}(\texttt{F}(f,ts),\texttt{V}(y))\ \} \tag{8.48}$$

$$\{\ \texttt{Lpo}(\texttt{V}(x),u) = \texttt{false}\ \} \tag{8.49}$$

$$\{\ \texttt{Alpha}(\texttt{nil},u) = \texttt{false}\ \} \tag{8.50}$$

$$\{\ \texttt{Alpha}(\texttt{cons}(t,ts),u) = \texttt{true},\ t \neq u\ \} \tag{8.51}$$

$$\{\ \texttt{Alpha}(\texttt{cons}(t,ts),u) = \texttt{true},\ t = u,\ \texttt{Lpo}(t,u) \neq \texttt{true}\ \} \tag{8.52}$$

$$\{\ \texttt{Alpha}(\texttt{cons}(t,ts),u) = \texttt{Alpha}(ts,u),\ t = u,\ \texttt{Lpo}(t,u) = \texttt{true},\ \neg\texttt{def Lpo}(t,u)\ \} \tag{8.53}$$

$$\{\ \texttt{Beta}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{Majo}(\texttt{F}(f,ts),us),\ \texttt{prec}(f,g) \neq \texttt{true}\ \} \tag{8.54}$$

$$\{\ \texttt{Beta}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{false},\ \texttt{prec}(f,g) = \texttt{true},\ \neg\texttt{def prec}(f,g)\ \} \tag{8.55}$$

$$\{\ \texttt{Gamma}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{Majo}(\texttt{F}(f,ts),us),\ f \neq g,\ \texttt{Lex}(ts,us) \neq \texttt{true}\ \} \tag{8.56}$$

$$\{\ \texttt{Gamma}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{false},\ f \neq g,\ \texttt{Lex}(ts,us) = \texttt{true},\ \neg\texttt{def Lex}(ts,us)\ \} \tag{8.57}$$

$$\{\ \texttt{Gamma}(\texttt{F}(f,ts),\texttt{F}(g,us)) = \texttt{false},\ f = g\ \} \tag{8.58}$$

$$\{\ \texttt{Delta}(\texttt{F}(f,ts),\texttt{V}(y)) = \texttt{contains\_tl}(ts,y)\ \} \tag{8.59}$$

$$\{\ \texttt{Majo}(t,\texttt{nil}) = \texttt{true}\ \} \tag{8.60}$$

$$\{\ \texttt{Majo}(t,\texttt{cons}(u,us)) = \texttt{Majo}(t,us),\ \texttt{Lpo}(t,u) \neq \texttt{true}\ \} \tag{8.61}$$

$$\{\ \texttt{Majo}(t,\texttt{cons}(u,us)) = \texttt{false},\ \texttt{Lpo}(t,u) = \texttt{true},\ \neg\texttt{def Lpo}(t,u)\ \} \tag{8.62}$$

$$\{\ \texttt{Lex}(\texttt{nil},\texttt{nil}) = \texttt{false}\ \} \tag{8.63}$$

$$\{\ \texttt{Lex}(\texttt{cons}(t,ts),\texttt{cons}(u,us)) = \texttt{Lex}(ts,us),\ t \neq u\ \} \tag{8.64}$$

$$\{\ \texttt{Lex}(\texttt{cons}(t,ts),\texttt{cons}(u,us)) = \texttt{Lpo}(t,u),\ t = u\ \} \tag{8.65}$$

The same transformations can be used for deriving the other variants from the specifications in [Löc04].

8.2.1.3 Auxiliary Lemmas and Formal Proofs

In this section, we describe the process of performing the formal proofs and finding suitable auxiliary lemmas without going into the details. These are presented in Section 8.2.2.

We started with an initial proof plan given by the proof sketches of the informal proofs. As properties of simplification orders such as transitivity and containedness of the subterm relation are omnipresent in the informal proofs, our first milestone was the verification of the property that `Lpo` defines a simplification order. For this, we proved some auxiliary lemmas about terms and the internal representation `Lpo`. The auxiliary lemmas originated from our domain knowledge as well as from the analysis of failed proof attempts. After having proved these basic properties, we proceeded by showing our main goal—the equivalence of the different variants as illustrated in Figure 8.3. During these proof attempts, we extended the data base of auxiliary lemmas if necessary.

The variants are organized in a hierarchy. The *main* variants are those variants described in [Löc04]. The *auxiliary* variants are introduced in the proof script e.g. to simplify the proofs for the main variants or to enable their definition. In the first case, the auxiliary variant is *defined by means of* another main variant such as `lpoR4` and `clpo4` which are both derived from `lpo4` (cf. Example 8.12). In this case, no optimizations—such as unfold/fold techniques—are applied to the auxiliary variant. It just converts the interface of the main variant. The second case applies to `lpoR6` which is required for the definition of `clpo6`.

Since we model the different variants of the LPO with boolean valued functions, equiv-

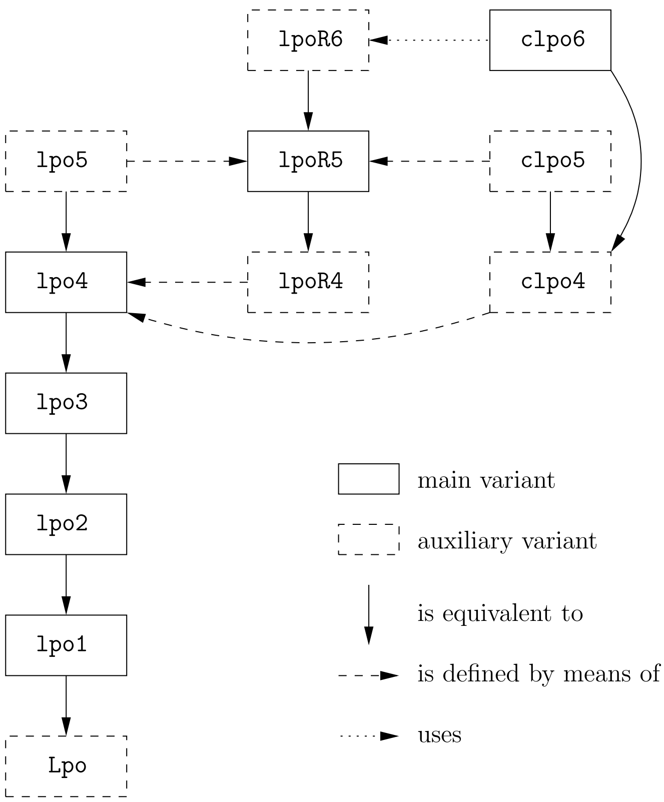

Figure 8.3: Relations Between the Different Variants of the LPO

alences between different variants are formulated as equations. We use these equations for rewriting from higher levels to lower levels. This is the reason for representing the equivalences as directed rewrite relations illustrated with arrows in Figure 8.3. The partiality of the operators is reflected in the specification of the equivalences: The equivalences hold true only for wellformed terms (cf. Examples 8.6 and 8.7).

The application of these equivalences as rewrite rules allows us to reduce the number of auxiliary lemmas dramatically: We specify auxiliary lemmas only for one variant, namely, the lowest variant w.r.t. the hierarchy illustrated in Figure 8.3 that allows for its formulation. Properties containing operator `lexM`, for instance, cannot be specified for variant `Lpo` but only for variants $\texttt{lpo}_i$ with $i \geq 3$ because the operator is introduced in variant $\texttt{lpo}_3$.[2] Properties between `alpha` and `majo`, however, are specified in variant `Lpo` using `Alpha` and `Majo` even if they are required only for variant `lpo3`. To apply the property in variant `lpo3`, we have to perform some more rewrite steps. But this approach allows us to formulate each property only once in the whole proof, and not for each variant separately.

The proofs of the equivalences are performed bottom-up w.r.t. the hierarchy in Figure 8.3. To simplify these proofs, we introduce auxiliary operators in one level that imitate the behavior of the new operators introduced in the next level. In doing so, we can isolate the proof of the the main property of the new operator from the proof of the equivalence relation between the two levels which has to be performed by mutual induction. In contrast to this, the isolated proof of the main property requires only simple induction. This will be explained in detail in Example 8.8.

We also faced the common problem of generalization in inductive theorem proving, namely, that lemmas cannot be proved by induction themselves but have to be generalized in such a way that an inductive proof can be performed. Then, the original lemma is proved by applying the generalized lemma. The generalization of a lemma may even require the introduction of new auxiliary operators (cf. Example 8.9).

The main challenge of the case study about the LPO is the use of mutual recursion in the definitions. Therefore, proofs have to be performed by mutual induction. If we want to prove a property for one operator defined by mutual recursion, appropriate lemmas for all dependent operators have to be specified and proved as well. Our lazy induction approach based on descente infinie (cf. Section 3.1.3) supports us in this endeavor. It allows us to provide the required information about the auxiliary lemmas, the inductive case split, and the induction order independently of each other and just when needed.

Note that our automatic proof control supports the proof of properties for mutually recursive operators only rudimentarily. The analysis process does not recognize mutual recursion but inspects each operator separately under the assumption that every other operator is terminating. This suffices to generate an inductive case split but other tasks have to be performed manually like

- the specification of auxiliary lemmas for the mutually recursive operators;
- the activation of the auxiliary lemmas for inductive applications;
- the instantiation of the weight variables to get an appropriate induction order.

[2] More precisely, we introduce `lexM` as an auxiliary operator for variant $\texttt{lpo}_2$ in Example 8.8 to simplify the equivalence proofs. Then, properties for `lexM` may also be specified for variant $\texttt{lpo}_2$.

Furthermore, the simplification process is not directed to use one special inductive instance of one of the mutually dependent lemmas as in explicit induction. This leads to many faulty inductive applications that have to be deleted again resulting in a less efficient simplification process because of the bigger search space. But it enables our proof control to find a proof at all with a less precise analysis of the operators. Therefore, our proof control succeeds in finding the proofs for the mutually recursive operators provided that the hints described above are given.

In the case study about the LPO, one of the most complicated steps in performing the mutual inductive proofs was the instantiation of the weight variables to get a suitable induction order. Recall that a weight variable may be instantiated with a tuple of terms containing the constructor variables of the goal to be proved. Weights—i.e. tuples of terms—are compared with a fixed wellfounded order, namely, the lexicographic order induced by the term lengths of the corresponding constructor terms (cf. Sections 2.2.1.1 and 2.2.3). Whereas the instantiation of the weight variables can be performed in a similar (and simple) way up to variant `lpo4`, it requires some ingenuity for the last two variants (cf. Examples 8.6 and 8.7). The last variant, for instance, swaps its arguments (cf. the definition of `clpo6` in Section A.17). Therefore, our lexicographic order is not really appropriate. Instead, the use of a multiset extension would be beneficial. We comment on such an extension in Section 8.3. For our case study, it was sufficient to instantiate the weight variables essentially with the sums of the lengths of the involved terms. Then, the swapping of the arguments can be compensated by using the commutativity of the addition. The integration of Hodes' decision procedures for linear arithmetic (cf. Chapter 4) supported us in this task.

Our first successful proof required more than 30000 seconds. Profiling the core system as well as the analysis of the statistics gathered during each proof attempt, revealed the reasons for some of these deficiencies:

- Within a tactic, inference rules may be called. If an inference rule is not applicable, its call will fail automatically. In the previous version of QML, this feature was extended to tactics themselves: The QML compiler generated code that determined whether the proof state tree had been modified at the end of an execution of a tactic. This information was used for throwing a failure automatically if the proof state tree did not change. Unfortunately, in general, this nice feature cannot be implemented efficiently in the presence of deletion operations which undo former applications of inference rules. Therefore, it has been abandoned in the new version of the core system. This causes a little more effort in the implementation of tactics because tests for determining a failure of a tactic have to be coded explicitly. However, efficiency increases dramatically since the specialized tests are often very simple.

- The analysis of the statistics revealed lemmas that were checked for applicability quite often but, in fact, the applications were never successful or at least only in very few cases. The use of obligatory markings—or even more drastic, the deactivation of lemmas—helped us to reduce unsuccessful checks for applicability (cf. Chapter 6).

  Another reason for failed proof attempts was the use of the strict mandatory markings heuristics for applicability subgoals (cf. Definition 6.6 in Section 6.2.1); proofs failed because of the restrictions caused by mandatory markings. Even worse, sometimes, these proofs were repeated for the subgoals again after another inference rule had been applied to the original goal. This observation resulted in the introduction of generous markings to relax the mandatory markings heuristics.

The use of a generous marking allows the automatic proof control to perform proof steps even if they do not contribute (locally). In general, the introduction of non-contributing proof steps decreases efficiency. This effect is compensated by our reuse mechanisms (cf. Chapter 7): Upward propagation restructures a performed proof by eliminating non-contributing proof steps. In doing so, it eliminates some open proof obligations.

In the end, we succeeded in reducing the runtime for the complete proof to nearly 250 seconds, a speed-up factor of 120.

The improvements described here as well as the proof techniques presented in Chapters 4 to 7 are not restricted to and have not been fine-tuned for this special case study. What we have learned here by analyzing the case study and improving our proof control is generally applicable as shown in the previous chapters.

8.2.2 The Resulting Proof Script: Problems and Particularities

The final proof script is divided into different modules: For each sort, there exists a separate file which contains the constructors, defined operators, and lemmas for these operators together with their proofs. There are modules for the boolean connectives, for terms and lists of terms as well as for each (main) variant of the LPO. In the following sections, we present some examples illustrating the problems in developing a proof script and their solutions in more detail.

8.2.2.1 Mutual Recursion/Induction

To get familiar with inductive reasoning over mutually recursive functions, we start with the simplest property of `Lpo`—its domain lemma. With this example we illustrate some of the most important steps that have to be performed manually when proving properties over mutually recursive operators with QuodLibet: the speculation of auxiliary lemmas and the instantiation of the weight variables to choose an appropriate induction order.

Example 8.4 At first, we have to speculate suitable domain lemmas for each of the operators `Lpo`, `Alpha`, `Beta`, `Gamma`, `Delta`, `Majo`, and `Lex` (cf. Axioms (8.45) to (8.65)). Note that `Lex` is defined only if the lists in both arguments have the same length. `Lex` is used only in the definition of `Gamma`. In this case, it is applied to the argument lists of two terms that start with the same function symbol. Since we consider function symbols with fixed arity only, we can guarantee the definedness property if both terms are *wellformed* (cf. Section 2.2.3). The restriction of wellformedness is then inherited to all other domain lemmas. Furthermore, `Beta` and `Gamma` are defined only if both argument terms start with a function symbol, whereas `Delta` is defined only if the first argument starts with a function symbol and the second argument term consists of a variable symbol. Therefore, we try to prove the domain Lemmas depicted in Figure 8.4 by mutual induction.

The *inductive dependencies* between the domain lemmas are illustrated in Figure 8.5. Lemmas (8.67), (8.68), and (8.69), for instance, are used as induction hypotheses in the

$$\{\ \texttt{def Lpo}(t,u),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true}\ \} \tag{8.66}$$

$$\{\ \texttt{def Alpha}(ts,t),\ \texttt{Well\_tl}(ts) \neq \texttt{true},\ \texttt{Well}(t) \neq \texttt{true}\ \} \tag{8.67}$$

$$\{\ \texttt{def Beta}(t,u),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true},\ \texttt{Fun}(t) \neq \texttt{true},\ \texttt{Fun}(u) \neq \texttt{true}\ \} \tag{8.68}$$

$$\{\ \texttt{def Gamma}(t,u),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true},\ \texttt{Fun}(t) \neq \texttt{true},\ \texttt{Fun}(u) \neq \texttt{true}\ \} \tag{8.69}$$

$$\{\ \texttt{def Delta}(t,u),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true},\ \texttt{Fun}(t) \neq \texttt{true},\ \texttt{Var}(u) \neq \texttt{true}\ \} \tag{8.70}$$

$$\{\ \texttt{def Majo}(t,us),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well\_tl}(us) \neq \texttt{true}\ \} \tag{8.71}$$

$$\{\ \texttt{def Lex}(ts,us),\ \texttt{length}(ts) \neq \texttt{length}(us),\ \texttt{Well\_tl}(ts) \neq \texttt{true},\ \texttt{Well\_tl}(us) \neq \texttt{true}\ \} \tag{8.72}$$

Figure 8.4: Domain Lemmas for Lpo

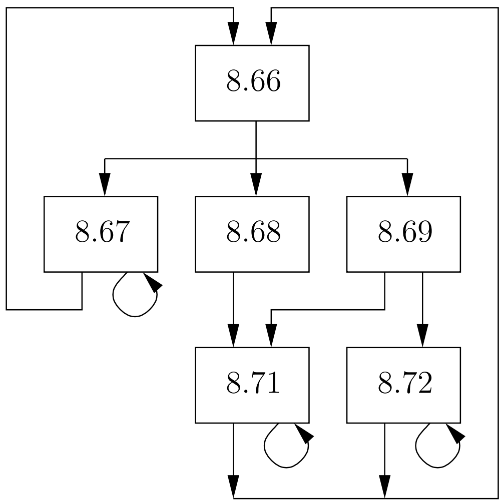

Figure 8.5: Inductive Dependencies Between the Domain Lemmas (8.66) to (8.72)

proof of Lemma (8.66). Note that the inductive dependencies in Figure 8.5 correspond to the recursive dependencies in the definition of the operators (cf. Figure 8.1).

There are different ways to express the properties that a term starts with a function symbol or consists only of a variable symbol: We may, for instance use, the corresponding constructor terms $\texttt{F}(f, ts)$ and $\texttt{V}(x)$, respectively. Then, the properties are guaranteed by matching operations. Instead, we use boolean valued operators `Fun` and `Var` defined as

$$\{\ \texttt{Fun}(\texttt{F}(f,ts)) = \texttt{true}\ \} \tag{8.73}$$

$$\{\ \texttt{Fun}(\texttt{V}(x)) = \texttt{false}\ \} \tag{8.74}$$

$$\{\ \texttt{Var}(\texttt{V}(x)) = \texttt{true}\ \} \tag{8.75}$$

$$\{\ \texttt{Var}(\texttt{F}(f,ts)) = \texttt{false}\ \} \tag{8.76}$$

$$w_{8.67}(\underline{ts}, \mathtt{F}(g, us)) < w_{8.66}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \tag{8.77}$$
$$w_{8.68}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) < w_{8.66}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \tag{8.78}$$
$$w_{8.69}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) < w_{8.66}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \tag{8.79}$$
$$w_{8.66}(\underline{u}, t) < w_{8.67}(\mathtt{cons}(u, ts), t) \tag{8.80}$$
$$w_{8.67}(\underline{ts}, t) < w_{8.67}(\mathtt{cons}(u, ts), t) \tag{8.81}$$
$$w_{8.71}(\mathtt{F}(f, ts), \underline{us}) < w_{8.68}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \tag{8.82}$$
$$w_{8.71}(\mathtt{F}(f, ts), \underline{us}) < w_{8.69}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \tag{8.83}$$
$$w_{8.72}(\underline{ts}, \underline{us}) < w_{8.69}(\mathtt{F}(f, ts), \mathtt{F}(g, us)) \tag{8.84}$$
$$w_{8.71}(t, \underline{us}) < w_{8.71}(t, \mathtt{cons}(u, us)) \tag{8.85}$$
$$w_{8.66}(t, \underline{u}) < w_{8.71}(t, \mathtt{cons}(u, us)) \tag{8.86}$$
$$w_{8.72}(\underline{ts}, \underline{us}) < w_{8.72}(\mathtt{cons}(t, ts), \mathtt{cons}(u, us)) \tag{8.87}$$
$$w_{8.66}(\underline{t}, \underline{u}) < w_{8.72}(\mathtt{cons}(t, ts), \mathtt{cons}(u, us)) \tag{8.88}$$

Figure 8.6: Order Constraints in the Proofs of the Domain Lemmas for `Lpo`

In doing so, the lemmas have additional conditions. On the one hand, these additional conditions reduce the efficiency of the proof process: If we apply Lemma (8.69) to a goal containing literal `def Gamma`$(\mathtt{F}(f, ts), \mathtt{F}(g, us))$ then we get two additional condition subgoals containing literal $\mathtt{Fun}(\mathtt{F}(f, ts)) = \mathtt{true}$ and $\mathtt{Fun}(\mathtt{F}(g, us)) = \mathtt{true}$, respectively. These condition subgoals can be proved immediately by applying Axiom (8.73). On the other hand, these lemmas are more often applicable. Lemma (8.69) can be applied additionally to a goal containing literals `def Gamma`$(t, \mathtt{F}(g, us))$ and $\mathtt{Fun}(t) \neq \mathtt{true}$.

If we perform the proofs of Lemmas (8.66) to (8.72) with our automatic proof control, the inductive applications of the lemmas result in order subgoals. Each order subgoal contains one of the order constraints depicted in Figure 8.6. We have to find a suitable instantiation of the weight variables such that every order subgoal can be proved. In general, we may use other literals of the order subgoals during their proofs as well. But in this case, the instantiated order constraints are sufficient to prove the order subgoals.

Recall that the weight variables have to be instantiated with tuples of terms which are compared with a lexicographic order based on the length of constructor terms (cf. Section 2.2.1). Thus, a constructor term t is smaller than a constructor term u if t is a strict subterm of u, i.e. a subterm which is unequal to u. Note that the constraints in Figure 8.6 contain different weight variables which may be instantiated in different ways. To cope with the resulting complexity, we ignore the different weight variables at first. Instead, we compare the corresponding arguments of the weight variables on the left-hand and right-hand side of the constraints as if we have instantiated each weight variable with the identity function. This approach is sensible since the arguments of the weight variables correspond to the arguments of the comparison operators `Lpo`, `Alpha`, `Beta`, `Gamma`, `Majo`, and `Lex` in the domain lemmas.

In Figure 8.6, we have underlined those arguments on the left-hand side which are definitely smaller than the arguments on the right-hand side as they are strict constructor subterms of the corresponding terms on the right-hand side. In Constraint (8.77) the second argument and in Constraint (8.82) the first argument remains the same. Therefore, both arguments are required to fulfill all order constraints. In the considered instantiation with the identity function, all constraints except for (8.78) and (8.79) are fulfilled. For these two constraints, we can exploit the fact that the weight variables on the left-hand and on the right-hand side are different. We may add a third component to the previous instantiation of $w_{8.66}$. This third component may consist of an arbitrary constructor term e.g. `0`. Then, all order constraints are fulfilled and the proofs can be completed. □

The next example illustrates the most complicated inductive proof performed within this case study: the proof of the transitivity of `Lpo` (cf. Lemma (8.89)) which mutually depends on the proof of the irreflexivity of `Lpo` (cf. Lemma (8.106)).

Example 8.5 We cannot describe the process in detail that has led to the specification of the 22 Lemmas (8.89) to (8.110) which express the transitivity and irreflexivity of the `Lpo` and the mutually dependent operators (cf. Figures 8.7 and 8.8). But the process was guided by the following considerations: We did not want to convert proofs of these properties from a textbook step-by-step. Instead, we wanted to exploit the automation of our proof control as far as possible. Therefore, most lemmas originated from the analysis of failed proof attempts. Certainly, we had to generalize the formulas of the failed proofs using our domain knowledge to derive suitable auxiliary lemmas. The proof attempt for Lemma (8.98) using our standard strategy without any lemmas activated for mutual induction, for instance, resulted in the following two open subgoals:

$$
\begin{array}{l}
\{\, \neg\texttt{def length}(us), \\
\quad \texttt{arity}(g) \neq \texttt{length}(us), \\
\quad \neg\texttt{def length}(vs), \\
\quad \texttt{arity}(g) \neq \texttt{length}(vs), \\
\quad \neg\texttt{def length}(ts), \\
\quad \neg\texttt{def arity}(g), \\
\quad \texttt{arity}(g) \neq \texttt{length}(ts), \\
\quad \texttt{Lex}(ts, vs) \neq \texttt{true}, \\
\quad \texttt{Lex}(us, vs) \neq \texttt{true}, \\
\quad \texttt{Lex}(ts, us) \neq \texttt{true}, \\
\quad \texttt{Majo}(\texttt{F}(g, ts), us) \neq \texttt{true}, \\
\quad \texttt{Majo}(\texttt{F}(g, us), vs) \neq \texttt{true}, \\
\quad \texttt{Majo}(\texttt{F}(g, ts), vs) = \texttt{true}, \\
\quad \texttt{Well\_tl}(ts) \neq \texttt{true}, \\
\quad \texttt{Well\_tl}(vs) \neq \texttt{true}, \\
\quad \texttt{Well\_tl}(us) \neq \texttt{true} \,\}
\end{array}
\tag{8.111}
$$

$$
\begin{array}{l}
\{\, \neg\texttt{def length}(us), \\
\quad \texttt{arity}(g) \neq \texttt{length}(us), \\
\quad \neg\texttt{def length}(vs), \\
\quad \texttt{arity}(g) \neq \texttt{length}(vs), \\
\quad \neg\texttt{def length}(ts), \\
\quad \neg\texttt{def arity}(g), \\
\quad \texttt{arity}(g) \neq \texttt{length}(ts), \\
\quad \neg\texttt{def Lex}(ts, vs), \\
\quad \texttt{Lex}(ts, vs) = \texttt{true}, \\
\quad \texttt{Lex}(us, vs) \neq \texttt{true}, \\
\quad \texttt{Lex}(ts, us) \neq \texttt{true}, \\
\quad \texttt{Majo}(\texttt{F}(g, ts), us) \neq \texttt{true}, \\
\quad \texttt{Majo}(\texttt{F}(g, us), vs) \neq \texttt{true}, \\
\quad \texttt{Well\_tl}(ts) \neq \texttt{true}, \\
\quad \texttt{Well\_tl}(vs) \neq \texttt{true}, \\
\quad \texttt{Well\_tl}(us) \neq \texttt{true} \,\}
\end{array}
\tag{8.112}
$$

Using our domain knowledge we derived the auxiliary Lemmas (8.101) and (8.102). We noticed, for instance, that Goal (8.111) generated by our automatic proof control remains true if nearly half of the literals is eliminated, resulting in Lemma (8.101).

Furthermore, some of the lemmas were introduced due to symmetry considerations. Lemmas (8.90) to (8.98), for instance, handle the nine cases where the comparisons in the

$$\{\, \mathtt{Lpo}(t,v) \neq \mathtt{true},\ \mathtt{Lpo}(v,u) \neq \mathtt{true},\ \mathtt{Lpo}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true} \,\} \tag{8.89}$$

$$\{\, \mathtt{Alpha}(ts,\mathtt{F}(h,vs)) \neq \mathtt{true},\ \mathtt{Alpha}(vs,u) \neq \mathtt{true},\ \mathtt{Alpha}(ts,u) = \mathtt{true},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well\_tl}(vs) \neq \mathtt{true},\ \mathtt{arity}(h) \neq \mathtt{length}(vs),\ \mathtt{Fun}(u) \neq \mathtt{true} \,\} \tag{8.90}$$

$$\{\, \mathtt{Alpha}(ts,v) \neq \mathtt{true},\ \mathtt{Beta}(v,u) \neq \mathtt{true},\ \mathtt{Alpha}(ts,u) = \mathtt{true},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true},\ \mathtt{Fun}(v) \neq \mathtt{true} \,\} \tag{8.91}$$

$$\{\, \mathtt{Alpha}(ts,v) \neq \mathtt{true},\ \mathtt{Gamma}(v,u) \neq \mathtt{true},\ \mathtt{Alpha}(ts,u) = \mathtt{true},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true},\ \mathtt{Fun}(v) \neq \mathtt{true} \,\} \tag{8.92}$$

$$\{\, \mathtt{Beta}(t,\mathtt{F}(h,vs)) \neq \mathtt{true},\ \mathtt{Alpha}(vs,u) \neq \mathtt{true},\ \mathtt{Lpo}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well\_tl}(vs) \neq \mathtt{true},\ \mathtt{arity}(h) \neq \mathtt{length}(vs),\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true} \,\} \tag{8.93}$$

$$\{\, \mathtt{Beta}(t,v) \neq \mathtt{true},\ \mathtt{Beta}(v,u) \neq \mathtt{true},\ \mathtt{Beta}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true},\ \mathtt{Fun}(v) \neq \mathtt{true} \,\} \tag{8.94}$$

$$\{\, \mathtt{Beta}(t,v) \neq \mathtt{true},\ \mathtt{Gamma}(v,u) \neq \mathtt{true},\ \mathtt{Beta}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true},\ \mathtt{Fun}(v) \neq \mathtt{true} \,\} \tag{8.95}$$

$$\{\, \mathtt{Gamma}(t,\mathtt{F}(h,vs)) \neq \mathtt{true},\ \mathtt{Alpha}(vs,u) \neq \mathtt{true},\ \mathtt{Lpo}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well\_tl}(vs) \neq \mathtt{true},\ \mathtt{arity}(h) \neq \mathtt{length}(vs),\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true} \,\} \tag{8.96}$$

$$\{\, \mathtt{Gamma}(t,v) \neq \mathtt{true},\ \mathtt{Beta}(v,u) \neq \mathtt{true},\ \mathtt{Beta}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true},\ \mathtt{Fun}(v) \neq \mathtt{true} \,\} \tag{8.97}$$

Figure 8.7: Auxiliary Lemmas for the Proof of the Transitivity of Lpo (1)

$$\{\ \mathtt{Gamma}(t,v) \neq \mathtt{true},\ \mathtt{Gamma}(v,u) \neq \mathtt{true},\ \mathtt{Gamma}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(v) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true},\ \mathtt{Fun}(v) \neq \mathtt{true}\ \} \tag{8.98}$$

$$\{\ \mathtt{Majo}(t,vs) \neq \mathtt{true},\ \mathtt{Alpha}(vs,u) \neq \mathtt{true},\ \mathtt{Lpo}(t,u) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well\_tl}(vs) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true}\ \} \tag{8.99}$$

$$\{\ \mathtt{Majo}(\mathtt{F}(f,ts),us) \neq \mathtt{true},\ \mathtt{Majo}(\mathtt{F}(g,us),vs) \neq \mathtt{true},\ \mathtt{Majo}(\mathtt{F}(f,ts),vs) = \mathtt{true},\ \mathtt{prec}(f,g) \neq \mathtt{true},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{arity}(f) \neq \mathtt{length}(ts),\ \mathtt{Well\_tl}(us) \neq \mathtt{true},\ \mathtt{arity}(g) \neq \mathtt{length}(us),\ \mathtt{Well\_tl}(vs) \neq \mathtt{true}\ \} \tag{8.100}$$

$$\{\ \mathtt{Majo}(\mathtt{F}(g,ts),us) \neq \mathtt{true},\ \mathtt{Majo}(\mathtt{F}(g,us),vs) \neq \mathtt{true},\ \mathtt{Majo}(\mathtt{F}(g,ts),vs) = \mathtt{true},\ \mathtt{Lex}(ts,us) \neq \mathtt{true},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{arity}(g) \neq \mathtt{length}(ts),\ \mathtt{Well\_tl}(us) \neq \mathtt{true},\ \mathtt{arity}(g) \neq \mathtt{length}(us),\ \mathtt{Well\_tl}(vs) \neq \mathtt{true}\ \} \tag{8.101}$$

$$\{\ \mathtt{Lex}(ts,us) \neq \mathtt{true},\ \mathtt{Lex}(us,vs) \neq \mathtt{true},\ \mathtt{Lex}(ts,vs) = \mathtt{true},\ \mathtt{length}(ts) \neq \mathtt{length}(us),\ \mathtt{length}(ts) \neq \mathtt{length}(vs),\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{Well\_tl}(vs) \neq \mathtt{true},\ \mathtt{Well\_tl}(us) \neq \mathtt{true}\ \} \tag{8.102}$$

$$\{\ \mathtt{Alpha}(ts,u) \neq \mathtt{true},\ \mathtt{Delta}(u,\mathtt{V}(y)) \neq \mathtt{true},\ \mathtt{Delta}(\mathtt{F}(f,ts),\mathtt{V}(y)) = \mathtt{true},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true},\ \mathtt{arity}(f) \neq \mathtt{length}(ts),\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(\mathtt{V}(y)) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true}\ \} \tag{8.103}$$

$$\{\ \mathtt{Beta}(t,u) \neq \mathtt{true},\ \mathtt{Delta}(u,\mathtt{V}(y)) \neq \mathtt{true},\ \mathtt{Delta}(t,\mathtt{V}(y)) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(\mathtt{V}(y)) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true}\ \} \tag{8.104}$$

$$\{\ \mathtt{Gamma}(t,u) \neq \mathtt{true},\ \mathtt{Delta}(u,\mathtt{V}(y)) \neq \mathtt{true},\ \mathtt{Delta}(t,\mathtt{V}(y)) = \mathtt{true},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Well}(u) \neq \mathtt{true},\ \mathtt{Well}(\mathtt{V}(y)) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true},\ \mathtt{Fun}(u) \neq \mathtt{true}\ \} \tag{8.105}$$

$$\{\ \mathtt{Lpo}(t,t) = \mathtt{false},\ \mathtt{Well}(t) \neq \mathtt{true}\ \} \tag{8.106}$$

$$\{\ \mathtt{Alpha}(ts,\mathtt{F}(g,us)) = \mathtt{false},\ \mathtt{sublist}(ts,us) \neq \mathtt{true},\ \mathtt{Well\_tl}(us) \neq \mathtt{true},\ \mathtt{arity}(g) \neq \mathtt{length}(us),\ \mathtt{Well\_tl}(ts) \neq \mathtt{true}\ \} \tag{8.107}$$

$$\{\ \mathtt{Beta}(t,t) = \mathtt{false},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true}\ \} \tag{8.108}$$

$$\{\ \mathtt{Gamma}(t,t) = \mathtt{false},\ \mathtt{Well}(t) \neq \mathtt{true},\ \mathtt{Fun}(t) \neq \mathtt{true}\ \} \tag{8.109}$$

$$\{\ \mathtt{Lex}(ts,ts) = \mathtt{false},\ \mathtt{Well\_tl}(ts) \neq \mathtt{true}\ \} \tag{8.110}$$

Figure 8.8: Auxiliary Lemmas for the Proof of the Transitivity of Lpo (2)

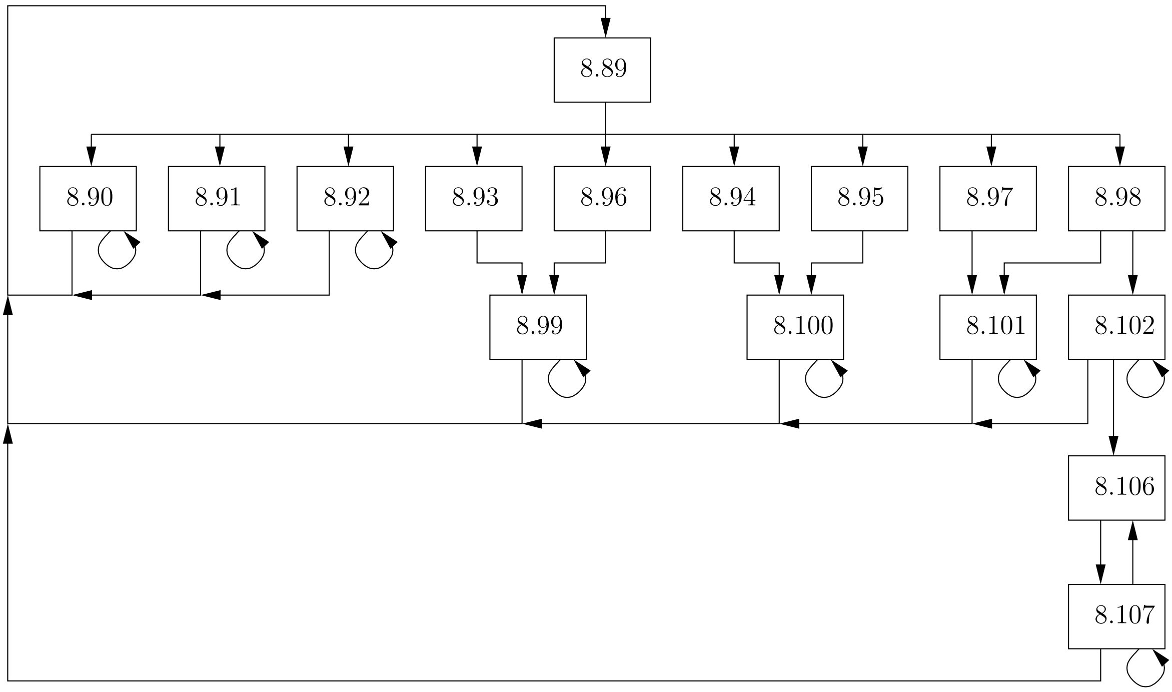

Figure 8.9: Inductive Dependencies in the Proof of the Transitivity of `Lpo`

premise of the transitivity are performed with one of the operators `Alpha`, `Beta`, and `Gamma`, respectively.

Lemmas (8.103), (8.104), (8.105), (8.108), (8.110), and (8.109) (using Lemma (8.110) non-inductively) can be proved in advance without mutual induction. The inductive dependencies between the remaining lemmas are illustrated in Figure 8.9.

The inductive applications in the proof attempts result in order subgoals which contain the 36 different order constraints presented in Figure 8.10. Since we have inductive dependencies between 16 lemmas, there are just as many weight variables that have to be instantiated. Therefore, there exist numerous different possibilities to instantiate them and many solutions that fulfill all the order constraints. We present one possible approach in doing this which extends the approach used in Example 8.4.

In Example 8.4, each domain lemma contained exactly two constructor variables (cf. Figure 8.4) which formed the arguments of the corresponding weight variables (cf. Figure 8.6). In the domain lemmas, these constructor variables were used as the arguments of the comparison operators. Therefore, it was sensible to ignore the different weight variables and to consider only their arguments at first.

The lemmas depicted in Figures 8.7 and 8.8, however, contain one to five constructor variables which serve different purposes. Therefore, it does not make sense to compare, for instance, the second argument of $w_{8.90}$, which is a single function symbol, with the second argument of $w_{8.89}$, which is a term. To apply the approach used in Example 8.4, we merge arguments of weight variables into *virtual* arguments. The merging process depends on the usage of the constructor variables in the corresponding lemmas. In Lemma (8.90), for instance, constructor variables h and vs form one single term $\mathtt{F}(h, vs)$ as second argument of `Alpha` in the first literal. Therefore, we merge the second and the third argument of

$$w_{8.90}(\underline{us}, \boxed{f, ts}, \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.113}$$
$$w_{8.91}(\underline{us}, \mathtt{F}(f, ts), \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.114}$$
$$w_{8.92}(\underline{us}, \mathtt{F}(f, ts), \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.115}$$
$$w_{8.93}(\mathtt{F}(g, us), \boxed{f, ts}, \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.116}$$
$$w_{8.94}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.117}$$
$$w_{8.95}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.118}$$
$$w_{8.96}(\mathtt{F}(g, us), \boxed{f, ts}, \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.119}$$
$$w_{8.97}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.120}$$
$$w_{8.98}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) < w_{8.89}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.121}$$
$$w_{8.89}(\underline{t}, \mathtt{F}(h, vs), u) < w_{8.90}(\mathtt{cons}(t, ts), \boxed{h, vs}, u) \tag{8.122}$$
$$w_{8.90}(\underline{ts}, \boxed{h, vs}, u) < w_{8.90}(\mathtt{cons}(t, ts), \boxed{h, vs}, u) \tag{8.123}$$
$$w_{8.89}(\underline{t}, v, u) < w_{8.91}(\mathtt{cons}(t, ts), v, u) \tag{8.124}$$
$$w_{8.91}(\underline{ts}, v, u) < w_{8.91}(\mathtt{cons}(t, ts), v, u) \tag{8.125}$$
$$w_{8.89}(\underline{t}, v, u) < w_{8.92}(\mathtt{cons}(t, ts), v, u) \tag{8.126}$$
$$w_{8.92}(\underline{ts}, v, u) < w_{8.92}(\mathtt{cons}(t, ts), v, u) \tag{8.127}$$
$$w_{8.99}(\mathtt{F}(f, ts), \underline{vs}, u) < w_{8.93}(\mathtt{F}(f, ts), \boxed{h, vs}, u) \tag{8.128}$$
$$w_{8.100}(\boxed{g, us}, \boxed{ts, f}, \underline{vs}) < w_{8.94}(\mathtt{F}(g, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.129}$$
$$w_{8.100}(\boxed{g, us}, \boxed{ts, h}, \underline{vs}) < w_{8.95}(\mathtt{F}(g, us), \mathtt{F}(h, ts), \mathtt{F}(h, vs)) \tag{8.130}$$
$$w_{8.99}(\mathtt{F}(f, ts), \underline{vs}, u) < w_{8.96}(\mathtt{F}(f, ts), \boxed{h, vs}, u) \tag{8.131}$$
$$w_{8.101}(\boxed{f, us}, \boxed{ts}, \underline{vs}) < w_{8.97}(\mathtt{F}(f, us), \mathtt{F}(f, ts), \mathtt{F}(h, vs)) \tag{8.132}$$
$$w_{8.101}(\boxed{h, us}, \boxed{ts}, \underline{vs}) < w_{8.98}(\mathtt{F}(h, us), \mathtt{F}(h, ts), \mathtt{F}(h, vs)) \tag{8.133}$$
$$w_{8.102}(\underline{us}, \underline{ts}, \underline{vs}) < w_{8.98}(\mathtt{F}(h, us), \mathtt{F}(h, ts), \mathtt{F}(h, vs)) \tag{8.134}$$
$$w_{8.89}(t, \underline{v}, u) < w_{8.99}(t, \mathtt{cons}(v, vs), u) \tag{8.135}$$
$$w_{8.99}(t, \underline{vs}, u) < w_{8.99}(t, \mathtt{cons}(v, vs), u) \tag{8.136}$$
$$w_{8.100}(\boxed{f, ts}, \boxed{us, g}, \underline{vs}) < w_{8.100}(\boxed{f, ts}, \boxed{us, g}, \mathtt{cons}(u, vs)) \tag{8.137}$$
$$w_{8.89}(\mathtt{F}(f, ts), \mathtt{F}(g, us), \underline{u}) < w_{8.100}(\boxed{f, ts}, \boxed{us, g}, \mathtt{cons}(u, vs)) \tag{8.138}$$
$$w_{8.101}(\boxed{g, ts}, \boxed{us}, \underline{vs}) < w_{8.101}(\boxed{g, ts}, \boxed{us}, \mathtt{cons}(u, vs)) \tag{8.139}$$
$$w_{8.89}(\mathtt{F}(g, ts), \mathtt{F}(g, us), \underline{u}) < w_{8.101}(\boxed{g, ts}, \boxed{us}, \mathtt{cons}(u, vs)) \tag{8.140}$$
$$w_{8.102}(\underline{ts}, \underline{us}, vs) < w_{8.102}(\mathtt{cons}(v, ts), \mathtt{cons}(v, us), \mathtt{cons}(v, vs)) \tag{8.141}$$
$$w_{8.89}(u, \underline{t}, \underline{v}) < w_{8.102}(\mathtt{cons}(u, ts), \mathtt{cons}(t, us), \mathtt{cons}(v, vs)) \tag{8.142}$$
$$w_{8.106}(\underline{v}) < w_{8.102}(\mathtt{cons}(v, ts), \mathtt{cons}(t, us), \mathtt{cons}(v, vs)) \tag{8.143}$$
$$w_{8.89}(\underline{v}, \underline{t}, \underline{v}) < w_{8.102}(\mathtt{cons}(v, ts), \mathtt{cons}(t, us), \mathtt{cons}(v, vs)) \tag{8.144}$$
$$w_{8.107}(\underline{ts}, \boxed{f, ts}) < w_{8.106}(\mathtt{F}(f, ts)) \tag{8.145}$$
$$w_{8.106}(\underline{u}) < w_{8.107}(\mathtt{cons}(u, ts), \boxed{g, us}) \tag{8.146}$$
$$w_{8.89}(\underline{u}, \mathtt{F}(g, us), u) < w_{8.107}(\mathtt{cons}(u, ts), \boxed{g, us}) \tag{8.147}$$
$$w_{8.107}(\underline{ts}, \boxed{g, us}) < w_{8.107}(\mathtt{cons}(u, ts), \boxed{g, us}) \tag{8.148}$$

Figure 8.10: Order Constraints in the Proof of the Transitivity of `Lpo`

the weight variable $w_{8.90}$ into one virtual argument. In Figure 8.10, virtual arguments that consist of more than one argument are framed. Note that in Lemma (8.101), function symbol g is used with argument lists ts and us. This is illustrated in Figure 8.10 by framing $\boxed{g, ts}$ and $\boxed{us}$ in $w_{8.101}$. Instead of the three arguments g, ts and us, we consider the two virtual arguments $\mathtt{F}(g, ts)$ and $\mathtt{F}(g, us)$. After this merging, $w_{8.106}$ has one, $w_{8.107}$ has two, and all other weight variables have three virtual arguments.

Now, we proceed as in Example 8.4, but compare the virtual arguments of the weight variables. In Figure 8.10, we have underlined those virtual arguments that are smaller w.r.t. the constructor term length than the corresponding virtual arguments on the other side of the constraint. Only the left-hand sides of the constraints contain smaller virtual arguments. If we instantiate each weight variable by the list of virtual arguments, all constraints except for (8.116) to (8.121) are fulfilled. These constraints can be fulfilled additionally if we add a fourth component such as $\mathtt{0}$ to the instantiation of $w_{8.89}$. We may simplify these instantiations by eliminating those components that are never used in the lexicographic comparison. In doing so, we derive the following instantiations:

$w_{8.89}(t, v, u) = (t, v, u, \mathtt{0})$	$w_{8.97}(t, v, u) = (t, v, u)$
$w_{8.90}(ts, h, vs, u) = ts$	$w_{8.98}(t, v, u) = (t, v, u)$
$w_{8.91}(ts, v, u) = ts$	$w_{8.99}(t, vs, u) = (t, vs)$
$w_{8.92}(ts, v, u) = ts$	$w_{8.100}(f, ts, us, g, vs) = (\mathtt{F}(f, ts), \mathtt{F}(g, us), vs)$
$w_{8.93}(t, h, vs, u) = (t, \mathtt{F}(h, vs))$	$w_{8.101}(g, ts, us, vs) = (\mathtt{F}(g, ts), \mathtt{F}(g, us), vs)$
$w_{8.94}(t, v, u) = (t, v, u)$	$w_{8.102}(ts, us, vs) = ts$
$w_{8.95}(t, v, u) = (t, v, u)$	$w_{8.106}(t) = t$
$w_{8.96}(t, h, vs, u) = (t, \mathtt{F}(h, vs))$	$w_{8.107}(ts, g, us) = ts$

With these instantiations, all order subgoals can be proved completing the proofs of the lemmas depicted in Figures 8.7 and 8.8.

This example illustrates the benefits of a proof process based on descente infinie for complicated inductive proofs. Both tasks, the speculation of the lemmas as well as the choice of the induction order are complicated on their own. With descente infinie, we are able to consider both tasks in isolation exploiting the information gathered so far, such as the order constraints for instantiating the weight variables. In contrast to this, in explicit induction all the information would have to be present right at the beginning of the proof attempt. □

The previous example suggests a structured approach for the instantiation of the weight variables. In principle, such an approach can be implemented as tactic to improve the automatic proof control. This is true for problems with a simple structure of mutual dependencies. But as the next two examples illustrate, the problems in finding a suitable instantiation of weight variables may become very hard in practice. In general, the problem whether there exists a suitable instantiation is undecidable. Therefore, we suppose that, often, this task still has to be done with human ingenuity in the future.

Example 8.6 In this example, we consider the equivalence proof between `lpoR5` and `lpoR4`. The defining rules of these variants can be found in Sections A.12 and A.13. The auxiliary lemmas that we have used for the proof are illustrated in Figure 8.11. Originally,

$$\{\ \texttt{lpoR5}(t,u) = \texttt{lpoR4}(t,u),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true}\ \} \tag{8.149}$$

$$\begin{aligned}\{\ & \texttt{lexMAE5}(\texttt{F}(f,vs),\texttt{F}(g,ws),ts,us) = \texttt{lexMAE4}(\texttt{F}(f,vs),\texttt{F}(g,ws),ts,us),\\ & \boxed{\texttt{sublist}(ts,vs) \neq \texttt{true}},\\ & \boxed{\texttt{sublist}(us,ws) \neq \texttt{true}},\\ & \texttt{arity}(f) \neq \texttt{length}(vs),\\ & \texttt{arity}(g) \neq \texttt{length}(ws),\\ & \texttt{length}(ts) \neq \texttt{length}(us),\\ & \texttt{Well\_tl}(vs) \neq \texttt{true},\\ & \texttt{Well\_tl}(ws) \neq \texttt{true},\\ & \texttt{Well\_tl}(ts) \neq \texttt{true},\\ & \texttt{Well\_tl}(us) \neq \texttt{true}\ \}\end{aligned} \tag{8.150}$$

$$\{\ \texttt{alphaR5}(ts,u) = \texttt{alphaR4}(ts,u),\ \texttt{Well\_tl}(ts) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true}\ \} \tag{8.151}$$

$$\{\ \texttt{majoR5}(t,us) = \texttt{majoR4}(t,us),\ \texttt{Well\_tl}(us) \neq \texttt{true},\ \texttt{Well}(t) \neq \texttt{true}\ \} \tag{8.152}$$

Figure 8.11: Auxiliary Lemmas for the Proof of the Equivalence Between `lpoR5` and `lpoR4`

$$w_{8.150}(\boxed{g,us},\boxed{g,us},\underline{us},\underline{us}) < w_{8.149}(\texttt{F}(g,us),\texttt{F}(g,us))^{*} \tag{8.153}$$
$$w_{8.152}(\texttt{F}(f,ts),\underline{us}) < w_{8.149}(\texttt{F}(f,ts),\texttt{F}(g,us)) \tag{8.154}$$
$$w_{8.150}(\boxed{g,ts},\boxed{g,us},\underline{ts},\underline{us}) < w_{8.149}(\texttt{F}(g,ts),\texttt{F}(g,us))^{*} \tag{8.155}$$
$$w_{8.151}(\underline{ts},\texttt{F}(g,us)) < w_{8.149}(\texttt{F}(f,ts),\texttt{F}(g,us)) \tag{8.156}$$

$$w_{8.150}(\boxed{f,vs},\boxed{g,ws},\underline{us},\underline{us}) < w_{8.150}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(u,us),\texttt{cons}(u,us)) \tag{8.157}$$
$$w_{8.149}(\underline{t},\underline{u}) < w_{8.150}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(t,ts),\texttt{cons}(u,us))^{**} \tag{8.158}$$
$$w_{8.150}(\boxed{f,vs},\boxed{g,ws},\underline{ts},\underline{us}) < w_{8.150}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(u,ts),\texttt{cons}(u,us)) \tag{8.159}$$
$$w_{8.152}(\texttt{F}(f,vs),\underline{us}) < w_{8.150}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(t,ts),\texttt{cons}(u,us))^{**} \tag{8.160}$$
$$w_{8.151}(\underline{ts},\texttt{F}(g,ws)) < w_{8.150}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(t,ts),\texttt{cons}(u,us))^{**} \tag{8.161}$$

$$w_{8.151}(\underline{ts},u) < w_{8.151}(\texttt{cons}(t,ts),u) \tag{8.162}$$
$$w_{8.149}(\underline{t},u) < w_{8.151}(\texttt{cons}(t,ts),u) \tag{8.163}$$

$$w_{8.152}(t,\underline{us}) < w_{8.152}(t,\texttt{cons}(u,us)) \tag{8.164}$$
$$w_{8.149}(t,\underline{u}) < w_{8.152}(t,\texttt{cons}(u,us)) \tag{8.165}$$

Figure 8.12: Order Constraints in the Proof of the Equivalence Between `lpoR5` and `lpoR4`

we tried to prove Lemma (8.150) without the framed literals. But for this stronger version we could not prove the order constraints using our fixed lexicographic order. The framed literals define a context in which the order constraints marked with two asterisks (**) in Figure 8.12 can be proved exploiting the following definition of `sublist`:

$$\{\ \mathtt{sublist}(\mathtt{nil}, us) = \mathtt{true}\ \} \tag{8.166}$$

$$\{\ \mathtt{sublist}(\mathtt{cons}(t, ts), \mathtt{nil}) = \mathtt{false}\ \} \tag{8.167}$$

$$\{\ \mathtt{sublist}(\mathtt{cons}(t, ts), \mathtt{cons}(u, us)) = \mathtt{sublist}(ts, us), \quad t \neq u\ \} \tag{8.168}$$

$$\{\ \mathtt{sublist}(\mathtt{cons}(t, ts), \mathtt{cons}(u, us)) = \mathtt{sublist}(\mathtt{cons}(t, ts), us), \quad t = u\ \} \tag{8.169}$$

More precisely, the inductive applications in the proof attempts of the lemmas illustrated in Figure 8.11 result in order subgoals containing the order constraints depicted in Figure 8.12. For the proofs of those order subgoals that contain order constraints marked with two asterisks, it is essential that they also contain the following literals:

$$\mathtt{sublist}(\mathtt{cons}(t, ts), vs) \neq \mathtt{true} \tag{8.170}$$

$$\mathtt{sublist}(\mathtt{cons}(u, us), ws) \neq \mathtt{true} \tag{8.171}$$

which are derived from the framed literals in Lemma (8.150).

To derive an instantiation of the weight variables, we use the same approach as in Example 8.5: We merge function symbols and argument lists into virtual arguments—illustrated by framing in Figure 8.12—according to their appearance as single terms in the corresponding lemmas in Figure 8.11. If we use these virtual arguments to instantiate the weight variables, the constraints that are not marked with an asterisk can be proved inductively valid because of the underlined arguments. These arguments are smaller than the corresponding ones on the right-hand side. Constraints marked with one asterisk hold true if we duplicate the arguments for the instantiation of $w_{8.149}$ (cf. below). Constraints marked with two asterisks can be proved inductively valid by exploiting Literals (8.170) and (8.171). In doing so, it suffices, for instance, to prove Constraint (8.158) under the condition that $\mathtt{cons}(t, ts)$ is a sublist of vs. But then, $t < \mathtt{F}(f, vs)$ holds true. Thus, we get the following instantiations of the weight variables:

$$\begin{aligned} w_{8.149}(t, u) &= (t, u, t, u) \\ w_{8.150}(f, vs, g, ws, ts, us) &= (\mathtt{F}(f, vs), \mathtt{F}(g, ws), ts, us) \\ w_{8.151}(ts, u) &= (ts, u) \\ w_{8.152}(t, us) &= (t, us) \end{aligned}$$

With these instantiations of the weight variables, the proofs of the lemmas depicted in Figure 8.11 can be completed with QuodLibet.

If we were allowed to use a multiset extension instead of the lexicographic order we could prove the strengthened version of Lemma (8.150) without the framed literals by using the same instantiations of the weight variables. This example illustrates that it may be beneficial to admit other wellfounded orders for performing the proofs of the order constraints. We comment on such an extension in Section 8.3.1. □

$$\{\ \texttt{clpo6}(t,u) = \texttt{clpo4}(t,u),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true}\ \} \tag{8.172}$$

$$\{\ \texttt{cLMA6}(\texttt{F}(f,vs),\texttt{F}(g,ws),ts,us) = \texttt{cLMA4}(\texttt{F}(f,vs),\texttt{F}(g,ws),ts,us),\ \boxed{\texttt{sublist}(ts,vs) \neq \texttt{true}},\ \boxed{\texttt{sublist}(us,ws) \neq \texttt{true}},\ \texttt{arity}(f) \neq \texttt{length}(vs),\ \texttt{arity}(g) \neq \texttt{length}(ws),\ \texttt{length}(ts) \neq \texttt{length}(us),\ \texttt{Well\_tl}(vs) \neq \texttt{true},\ \texttt{Well\_tl}(ws) \neq \texttt{true},\ \texttt{Well\_tl}(ts) \neq \texttt{true},\ \texttt{Well\_tl}(us) \neq \texttt{true}\ \} \tag{8.173}$$

$$\{\ \texttt{cMA6}(t,us) = \texttt{cMA4}(t,us),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well\_tl}(us) \neq \texttt{true}\ \} \tag{8.174}$$

$$\{\ \texttt{cAA6}(t,u,ts,us) = \texttt{cAA4}(t,u,ts,us),\ \texttt{Well}(t) \neq \texttt{true},\ \texttt{Well}(u) \neq \texttt{true},\ \texttt{Well\_tl}(ts) \neq \texttt{true},\ \texttt{Well\_tl}(us) \neq \texttt{true}\ \} \tag{8.175}$$

Figure 8.13: Auxiliary Lemmas for the Proof of the Equivalence Between `clpo6` and `clpo4`

$$w_{8.173}(\boxed{g,us},\boxed{g,us},\underline{us},\underline{us}) < w_{8.172}(\texttt{F}(g,us),\texttt{F}(g,us))^{*} \tag{8.176}$$

$$w_{8.173}(\boxed{g,ts},\boxed{g,us},\underline{ts},\underline{us}) < w_{8.172}(\texttt{F}(g,ts),\texttt{F}(g,us))^{*} \tag{8.177}$$

$$w_{8.174}(\texttt{F}(f,ts),\underline{us}) < w_{8.172}(\texttt{F}(f,ts),\texttt{F}(g,us)) \tag{8.178}$$

$$w_{8.174}(\texttt{F}(g,us),\underline{ts}) < w_{8.172}(\texttt{F}(f,ts),\texttt{F}(g,us))^{***} \tag{8.179}$$

$$w_{8.173}(\boxed{f,vs},\boxed{g,ws},\underline{us},\underline{us}) < w_{8.173}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(ui,us),\texttt{cons}(ui,us)) \tag{8.180}$$

$$w_{8.172}(\underline{ti},\underline{ui}) < w_{8.173}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(ti,ts),\texttt{cons}(ui,us))^{**} \tag{8.181}$$

$$w_{8.173}(\boxed{f,vs},\boxed{g,ws},\underline{ts},\underline{us}) < w_{8.173}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(ui,ts),\texttt{cons}(ui,us)) \tag{8.182}$$

$$w_{8.174}(\texttt{F}(f,vs),\underline{us}) < w_{8.173}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(ti,ts),\texttt{cons}(ui,us))^{**} \tag{8.183}$$

$$w_{8.174}(\texttt{F}(g,ws),\underline{ts}) < w_{8.173}(\boxed{f,vs},\boxed{g,ws},\texttt{cons}(ti,ts),\texttt{cons}(ui,us))^{**,***} \tag{8.184}$$

$$w_{8.174}(t,\underline{us}) < w_{8.174}(t,\texttt{cons}(u,us)) \tag{8.185}$$

$$w_{8.172}(t,\underline{u}) < w_{8.174}(t,\texttt{cons}(u,us)) \tag{8.186}$$

Figure 8.14: Order Constraints in the Proof of the Equivalence Between `clpo6` and `clpo4`

Example 8.7 This example is similar to but still a little bit more complicated than Example 8.6. It illustrates the equivalence proof between `clpo6` and `clpo4` (cf. Sections A.15 and A.17 for their definitions). The additional problem is that during the bidirectional comparison in `clpo6`, arguments are swapped in the recursive calls occurring in the defining rules.

Figure 8.13 contains the auxiliary lemmas used for the proof. As for Lemma (8.150) in Example 8.6, the framed literals in Lemma (8.173) are present just to facilitate the proof of those order subgoals that contain order constraints marked with two asterisks in Figure 8.14. In these order subgoals the framed literals are instantiated to

$$\mathtt{sublist}(\mathtt{cons}(ti, ts), vs) \neq \mathtt{true} \tag{8.187}$$
$$\mathtt{sublist}(\mathtt{cons}(ui, us), ws) \neq \mathtt{true} \tag{8.188}$$

The definition of `cAA6` does not depend on the other three operators. Therefore, the proof can be performed by simple induction. The inductive applications in the proofs of the remaining lemmas of Figure 8.13 result in order subgoals which contain the order constraints depicted in Figure 8.14. The figure is annotated by framing and underlining as well as with asterisks in the same way as Figure 8.12 in Example 8.6:

- arguments are merged into virtual arguments by framing;
- virtual arguments in the left-hand side that are smaller than the corresponding arguments in the right-hand are underlined;[3]
- for the instantiation of $w_{8.172}$, we duplicate its arguments due to the order constraints marked with one asterisk;
- the proofs of the order constraints marked with two asterisks exploit Literals (8.187) and (8.188) which are contained in the corresponding order subgoals;
- the order constraints marked with three asterisks contain weight variables with swapped arguments. To be able to prove these constraints, we combine the first two arguments with a commutative operator in all the instantiations of the weight variables. This last item is new in comparison to Example 8.6.

Therefore, we get the following instantiations for the weight variables which allow us to prove the order constraints in Figure 8.14 with our lexicographic order:

$$\begin{aligned}
w_{8.172}(t, u) &= (\mathtt{+}(\mathtt{term\text{-}size}(t), \mathtt{term\text{-}size}(u)),\\
&\qquad \mathtt{term\text{-}size}(t))\\
w_{8.173}(f, vs, g, ws, ts, us) &= (\mathtt{+}(\mathtt{term\text{-}size}(\mathtt{F}(f, vs)), \mathtt{term\text{-}size}(\mathtt{F}(g, ws))),\\
&\qquad \mathtt{term\text{-}size_tl}(ts))\\
w_{8.174}(t, us) &= \mathtt{+}(\mathtt{term\text{-}size}(t), \mathtt{term\text{-}size_tl}(us))
\end{aligned}$$

The auxiliary operators `term-size` and `term-size_tl` measure the size of terms and lists of terms. We define the size of a term similarly to its length except that for each argument list

[3] taking into account the instantiation of the weight variables according to the asterisks

the size is additionally incremented by 1. This simplifies the proofs of the order subgoals. We get the following definitions:

$$\{ \texttt{term-size}(\texttt{V}(x)) = \texttt{1} \} \tag{8.189}$$
$$\{ \texttt{term-size}(\texttt{F}(f, ts)) = \texttt{s}(\texttt{term-size\_tl}(ts)) \} \tag{8.190}$$

$$\{ \texttt{term-size\_tl}(\texttt{nil}) = \texttt{1} \} \tag{8.191}$$
$$\{ \texttt{term-size\_tl}(\texttt{cons}(t, ts)) = \texttt{+}(\texttt{term-size}(t), \texttt{term-size\_tl}(ts)) \} \tag{8.192}$$

Hodes' decision procedure for linear arithmetic (cf. Section 4) helps us to prove the order subgoals with QUODLIBET by exploiting the commutativity of the addition.

Once again, using a multiset extension instead of the lexicographic order as fixed well-founded order for comparing the order constraints would allow us to prove the strengthened version of Lemma (8.173) without the framed literals and to simplify the proofs of the order subgoals using the following instantiations of the weight variables:

$$\begin{aligned} w_{8.172}(t, u) &= (t, u, t, u) \\ w_{8.173}(f, vs, g, ws, ts, us) &= (\texttt{F}(f, vs), \texttt{F}(g, ws), ts, us) \\ w_{8.174}(t, us) &= (t, us) \end{aligned}$$

We will comment on such an extension in Section 8.3.1. □

With the last example in this section, we concretize the use of auxiliary operators to simplify the mutually inductive proofs of the equivalences between the different levels.

Example 8.8 In variant `lpo3` a new operator `lexM3` is defined which replaces the calls to `lex` and `majo` in `gamma3` to avoid the redundant calls to `majo` for $k \leq i$ in Definition 8.1. We have the following definitions:

$$\{ \texttt{gamma3}(\texttt{F}(f, ts), \texttt{F}(g, us)) = \texttt{lexM3}(\texttt{F}(f, ts), ts, us), \; f \neq g \} \tag{8.193}$$
$$\{ \texttt{gamma3}(\texttt{F}(f, ts), \texttt{F}(g, us)) = \texttt{false}, \; f = g \} \tag{8.194}$$

$$\{ \texttt{lexM3}(v, \texttt{nil}, \texttt{nil}) = \texttt{false} \} \tag{8.195}$$
$$\{ \texttt{lexM3}(v, \texttt{cons}(t, ts), \texttt{cons}(u, us)) = \texttt{lexM3}(v, ts, us), \; t \neq u \} \tag{8.196}$$
$$\{ \texttt{lexM3}(v, \texttt{cons}(t, ts), \texttt{cons}(u, us)) = \texttt{and}(\texttt{lpo3}(t, u), \texttt{majo3}(v, us)), \; t = u \} \tag{8.197}$$

In contrast to this, `gamma2` is defined with the following axioms:

$$\{ \texttt{gamma2}(\texttt{F}(f, ts), \texttt{F}(g, us)) = \texttt{and}(\texttt{lex2}(ts, us), \texttt{majo2}(\texttt{F}(f, ts), us)), \; f \neq g \} \tag{8.198}$$
$$\{ \texttt{gamma2}(\texttt{F}(f, ts), \texttt{F}(g, us)) = \texttt{false}, \; f = g \} \tag{8.199}$$

If we want to prove the equivalence between the two variants $\texttt{lpo}_2$ and $\texttt{lpo}_3$, we have to verify among other things:

$$\begin{aligned}\{\ & \texttt{gamma3}(t,u) = \texttt{gamma2}(t,u),\\ & \texttt{Well}(t) \neq \texttt{true},\\ & \texttt{Well}(u) \neq \texttt{true},\\ & \texttt{Fun}(t) \neq \texttt{true},\\ & \texttt{Fun}(u) \neq \texttt{true}\ \}\end{aligned} \tag{8.200}$$

For this, we could try to prove:

$$\begin{aligned}\{\ & \texttt{lexM3}(\texttt{F}(f,ts),ts,us) = \texttt{and}(\texttt{lex2}(ts,us),\texttt{majo2}(\texttt{F}(f,ts),us)),\\ & \texttt{length}(ts) \neq \texttt{length}(us),\\ & \texttt{arity}(f) \neq \texttt{length}(ts),\\ & \texttt{Well\_tl}(ts) \neq \texttt{true},\\ & \texttt{Well\_tl}(us) \neq \texttt{true}\ \}\end{aligned} \tag{8.201}$$

But then, the mutual inductive proofs of the equivalence would become more difficult. In fact, we could not prove Lemma (8.201) immediately by induction but we would have to generalize it first (cf. Example 8.9).

Since we want to facilitate the mutual inductive proofs as far as possible, we prefer to decouple the proof of the main property of the new operator `lexM` from the proof of the equivalence relation. For this, we introduce a corresponding operator `lexM2` for variant `lpo2` with the following axioms:

$$\{\ \texttt{lexM2}(t,\texttt{nil},\texttt{nil}) = \texttt{false}\ \} \tag{8.202}$$

$$\begin{aligned}\{\ & \texttt{lexM2}(v,\texttt{cons}(t,ts),\texttt{cons}(u,us)) = \texttt{lexM2}(v,ts,us),\\ & t \neq u\ \}\end{aligned} \tag{8.203}$$

$$\begin{aligned}\{\ & \texttt{lexM2}(v,\texttt{cons}(t,ts),\texttt{cons}(u,us)) = \texttt{and}(\texttt{lpo2}(t,u),\texttt{majo2}(v,us)),\\ & t = u\ \}\end{aligned} \tag{8.204}$$

Then, we prove the main property of `lexM` for variant `lpo2`, namely:

$$\begin{aligned}\{\ & \texttt{lexM2}(\texttt{F}(f,ts),ts,us) = \texttt{and}(\texttt{Lex}(ts,us),\texttt{Majo}(\texttt{F}(f,ts),us)),\\ & \texttt{length}(ts) \neq \texttt{length}(us),\\ & \texttt{arity}(f) \neq \texttt{length}(ts),\\ & \texttt{Well\_tl}(ts) \neq \texttt{true},\\ & \texttt{Well\_tl}(us) \neq \texttt{true}\ \}\end{aligned} \tag{8.205}$$

Just like Lemma (8.201), Lemma (8.205) cannot be proved immediately by induction but we have to generalize it first. But then, its generalization (8.207) can be proved by *simple* induction instead of mutual induction (cf. Example 8.9).

With Lemma (8.205), we can prove Lemma (8.200) as well as the following lemma easily by mutual induction:

$$\begin{aligned}\{\ & \texttt{lexM3}(v,ts,us) = \texttt{lexM2}(v,ts,us),\\ & \texttt{length}(ts) \neq \texttt{length}(us),\\ & \texttt{Well}(v) \neq \texttt{true},\\ & \texttt{Well\_tl}(ts) \neq \texttt{true},\\ & \texttt{Well\_tl}(us) \neq \texttt{true}\ \}\end{aligned} \tag{8.206}$$

The proof depends on further lemmas for `lpo3`, `alpha3`, `beta3`, and `majo3`. □

8.2.2.2 Speculation of Auxiliary Lemmas

In inductive theorem proving, the application of lemmas is very important for performing proofs *successfully* and *efficiently*. Therefore, the speculation of appropriate auxiliary lemmas as well as the integration of the lemma application mechanism into the automatic proof control is essential. In this section, we concentrate on the speculation of auxiliary lemmas. The efficiency of proof search is considered in Section 8.2.2.3.

Often, a failed proof attempt calls for an additional lemma and the analysis of one of the open goal nodes results in an appropriate candidate. The final version of a useful auxiliary lemma usually requires some kind of generalization (cf. Example 8.5). A lemma may be generalized by eliminating some of its premises, i.e. literals in its clause, or by replacing common subterms in the lemma with new variables. The following example illustrates that the derivation of suitable generalizations is sometimes very difficult. In these cases, we have to perform this task manually.

Example 8.9 In Example 8.8, we have identified Lemma (8.205) as an appropriate auxiliary lemma. But this lemma cannot be proved by induction immediately: An inductive proof would have to be performed on the second and third argument of operator `lexM2`, namely on the variables *ts* and *us*. But as *ts* also occurs in the first argument, none of the induction hypotheses would be applicable.

This problem can be solved by separating the two "occurrences" of *ts*, i.e. one "occurrence" is replaced *uniformly* with a new variable *ts1*. This means that we check for each individual appearance of *ts* to which "occurrence" it refers, possibly replacing it with the new variable *ts1*. This results in the following lemma which can be proved by induction:

$$\begin{aligned}\{\ & \texttt{lexM2}(\texttt{F}(f, ts1), ts, us) = \texttt{and}(\texttt{Lex}(ts, us), \texttt{Majo}(\texttt{F}(f, ts1), us)),\\ & \texttt{sublist}(ts, ts1) \neq \texttt{true},\\ & \texttt{arity}(f) \neq \texttt{length}(ts1),\\ & \texttt{Well\_tl}(ts1) \neq \texttt{true},\\ & \texttt{Well\_tl}(ts) \neq \texttt{true},\\ & \texttt{Well\_tl}(us) \neq \texttt{true},\\ & \texttt{length}(ts) \neq \texttt{length}(us)\ \}\end{aligned} \tag{8.207}$$

The inductive validity of this lemma depends on the second literal. It manifests the relationship between the two "occurrences" *ts* and *ts1* using the *new* operator `sublist` which does not appear in Lemma (8.205) (cf. Axioms 8.166 to 8.169 in Example 8.6 for the definition of `sublist`).

For these two reasons, it is difficult to speculate Lemma (8.207) automatically: Firstly, we replace only *some* individual appearances of variable *ts* with the new variable *ts1*. This is just in the realm of syntactic automatization as found e.g. in `NQTHM`. Secondly, the relationship between both occurrences has to be expressed using a new operator symbol. This requires semantic knowledge and would be possible only if a special treatment for list operations was built in. In general, however, for each new application domain, human interaction for lemma generalization will be required.

Although Lemma (8.207) is a generalization of Lemma (8.205), it is beneficial to prove Lemma (8.205) as well by applying Lemma (8.207). Thereafter, Lemma (8.205) may be activated instead of Lemma (8.207) to reduce the search space. □

In addition to patching proofs, additional auxiliary lemmas may also be useful for enhancing the efficiency of the proofs performed. This is illustrated in Example 8.10.

8.2.2.3 Improving the Efficiency of Proof Search

In this section, we demonstrate the benefits of our flexible framework for guiding proof search with markings (cf. Chapter 6). In particular, we illustrate the usage of generous markings and its interplay with upward propagation (cf. Chapter 7) in Examples 8.11 and 8.12.

Primarily, the efficiency of proof search in inductive theorem provers such as QuodLibet, `NQTHM` and `ACL2` depends on the lemmas that may be applied during proof search. In `NQTHM` and `ACL2`, only activated lemmas are considered for automatic applications. Furthermore, the applications of conditional lemmas are controlled by Contextual Rewriting. Our new framework based on markings allows us to influence the automatic application of activated lemmas even more. Our default heuristics with mandatory markings restricts proof search based on the idea of local contribution. With obligatory markings, we may restrict lemma applications furthermore. With generous markings, we may relax the restrictions of our default heuristics.

Sometimes, proof attempts fail (cf. Chapter 6) or proof search lacks efficiency (cf. Example 8.10) just because of the restrictions on lemma applications. In our flexible framework, we may cope with these problems by

- introducing specialized additional auxiliary lemmas (cf. Example 8.10); or
- changing the behavior of proof search with markings (cf. Examples 8.11).

Usually, the use of specialized auxiliary lemmas improves the efficiency of proof search even more but it requires additional human effort for formulating or speculating suitable lemmas.

Example 8.10 In the proof of Lemma (8.66), the following subgoal is generated:

$$\begin{aligned}\{\ & \texttt{def Alpha}(ts, \texttt{F}(g, us)),\\ & \texttt{Beta}(\texttt{F}(f, ts), \texttt{F}(g, us)) \neq \texttt{true},\\ & \texttt{Alpha}(ts, \texttt{F}(g, us)) = \texttt{true},\\ & \texttt{def Lpo}(\texttt{F}(f, ts), \texttt{F}(g, us)),\\ & \texttt{Well}(\texttt{F}(f, ts)) \neq \texttt{true},\\ & \texttt{Well}(\texttt{F}(g, us)) \neq \texttt{true}\ \}\end{aligned} \tag{8.208}$$

In this example, we consider various proof attempts for this goal. At first, we assume the activation of the axioms and lemmas for the involved operators as presented in Section 2.2.3 and the previous sections of this chapter. Furthermore, we assume the activation of the following domain lemma for the precedence relation `prec`:

$$\{\ \texttt{def prec}(f, g)\ \} \tag{8.209}$$

The first proof attempt of the automatic proof control is sketched in Figure 8.15. Mandatory literals are framed, principal literals are underlined (cf. Chapters 5 and 6). Whereas the

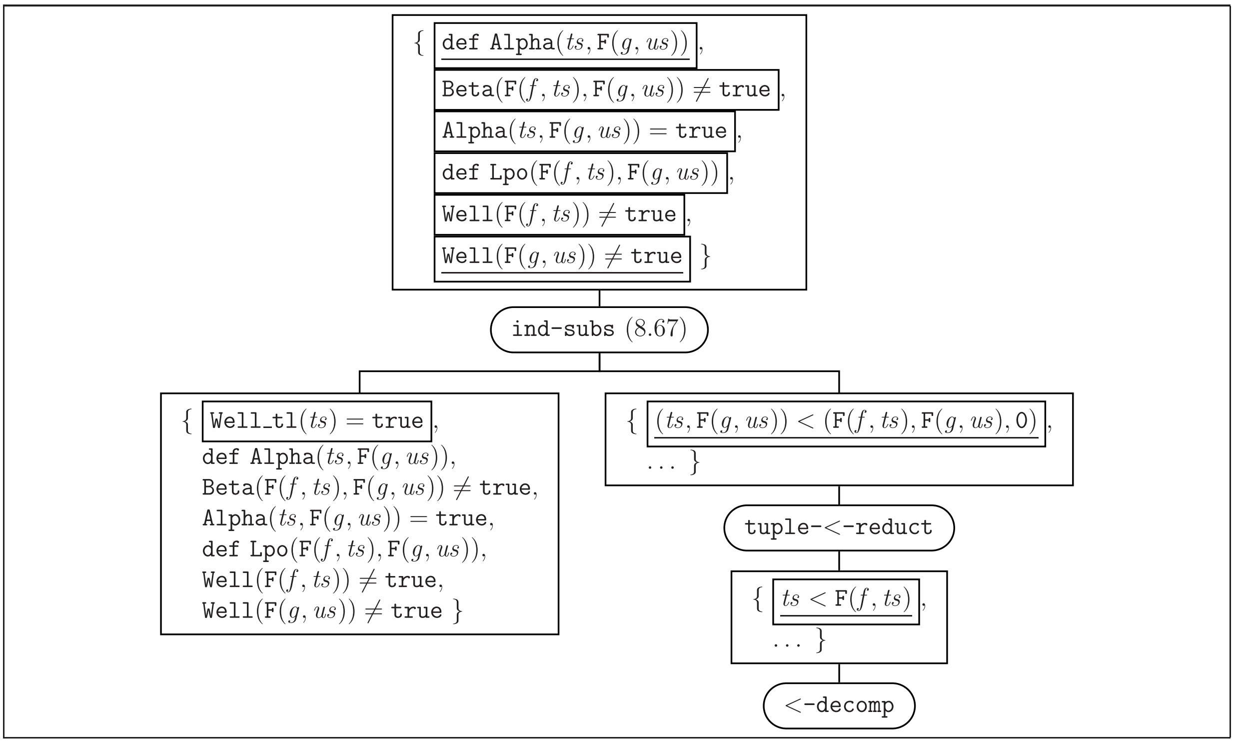

Figure 8.15: First Proof Attempt for Goal (8.208) of Example 8.10

proof of the order subgoal—the second one—succeeds, the conditional subgoal cannot be proved with the present axioms and lemmas due to the restrictions caused by the mandatory marking. To be more precise, the condition subgoal contains the following subformula

$$\begin{aligned} \{ \; & \texttt{Well}(\texttt{F}(f, ts)) \neq \texttt{true}, \\ & \texttt{Well\_tl}(ts) = \texttt{true} \; \} \end{aligned} \tag{8.210}$$

Goal (8.210) can be proved easily as illustrated in Figure 8.16. But the first proof step depends only on the first literal. Therefore, it cannot be applied to the condition subgoal in Figure 8.15 because of the mandatory marking.

Instead, the proof attempt fails and another one is started automatically as sketched in Figure 8.17. In fact, this proof attempt succeeds by inverting the order in which Lemma (8.67) and Axiom (2.5) are applied. Additionally, the proof attempt contains many unnecessary applications of further axioms and lemmas. Thus, it is very inefficient.

In contrast to this, the first proof attempt for Goal (8.208) as illustrated in Figure 8.15 can be closed immediately if we use Lemma (8.210) as auxiliary lemma. The activation of this lemma guides proof search in the right direction: It eliminates the unsuccessful proof attempt and avoids unnecessary applications of inference rules.

In fact, we prefer to formulate lemmas as rewrite rules if possible. Therefore, we have replaced Lemma (8.210) with

$$\begin{aligned} \{ \; & \texttt{Well}(\texttt{F}(f, ts)) = \texttt{false}, \\ & \texttt{Well\_tl}(ts) = \texttt{true} \; \} \end{aligned} \tag{8.211}$$

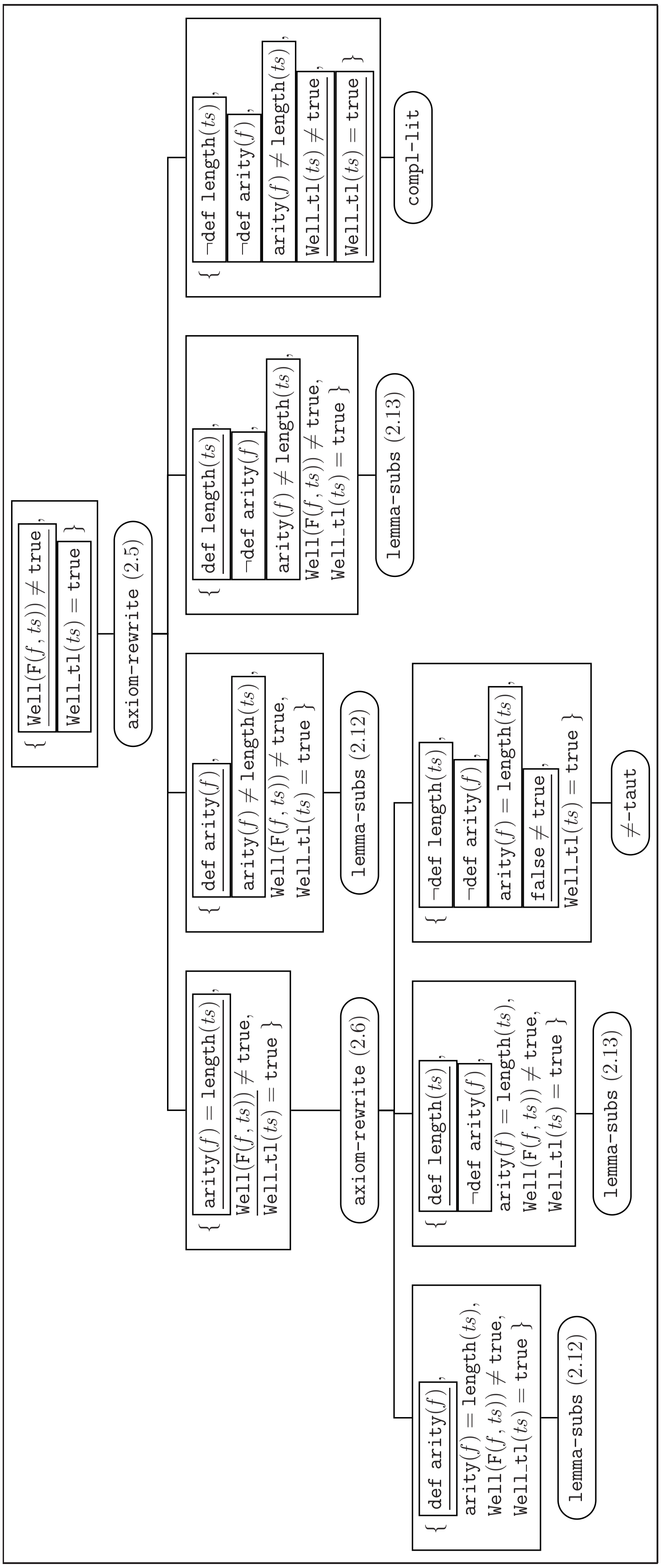

Figure 8.16: Proof State Tree for Goal (8.210) of Example 8.10

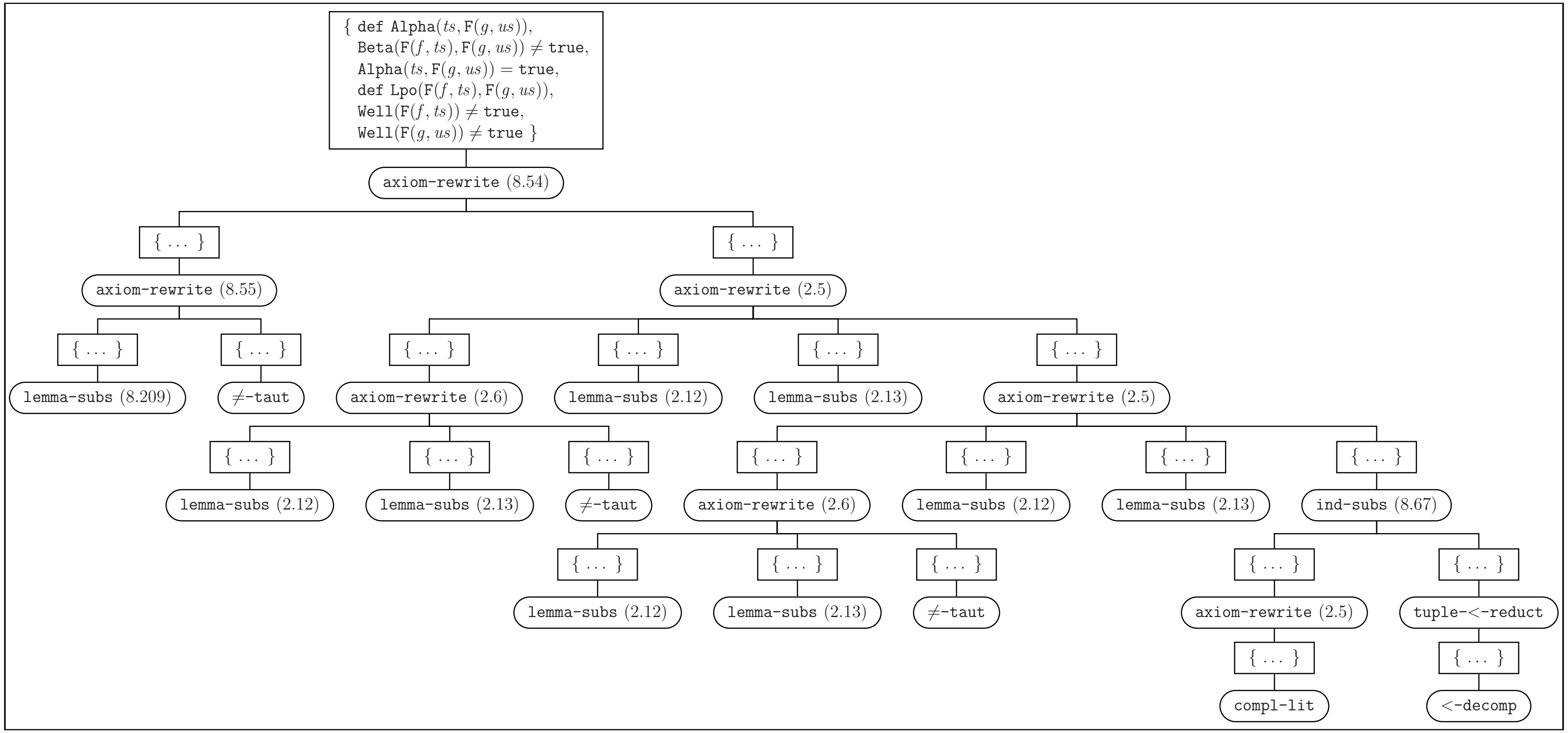

Figure 8.17: Second Proof Attempt for Goal (8.208) of Example 8.10

Lemmas (8.211) and (8.212)	Autom. Appl.	Del.	Fin. P.	Runtime
not activated	4329	1028	3301	43.66
activated	2722	253	2469	20.93

Table 8.1: Statistics for the 66 Lemmas about `Lpo` w.r.t. the Activation of Lemmas

In the same way, we have identified a similar auxiliary lemma:

$$\begin{aligned}\{\ &\texttt{Well\_tl}(\texttt{cons}(t, ts)) = \texttt{false},\\ &\texttt{Well\_tl}(ts) = \texttt{true}\ \}\end{aligned} \tag{8.212}$$

The activation of these two lemmas results in dramatic improvements in efficiency. This is illustrated for the 66 lemmas about the internal representation `Lpo` in Table 8.1. □

The analysis of proof attempts for patching failed ones or improving the efficiency of successful ones may be time-consuming. Due to our case study about the LPO, we have introduced a new concept—generous markings—for influencing our automatic proof control manually.

Generous markings can be used for relaxing the mandatory markings heuristics (cf. Section 6.2.4). Thus, generous markings enhance the extent of the relief test. But they may also enhance the efficiency provided that they are used with caution. Generous markings are intended to relax the mandatory markings heuristics for those subgoals that are expected to be proved easily. Usually, this holds true for subgoals that contain new definedness atoms. Therefore, our proof control offers a new default setting. If this setting is activated, a suitable generous marking w.r.t. definedness atoms is accomplished automatically. In this case, definedness subgoals are generated with the relaxed mandatory markings heuristics. Furthermore, in all lemmas, negated definedness atoms—which generate definedness atoms in the corresponding condition subgoals—and, in domain lemmas, all literals are marked as generous by default. In doing so, elements remain mandatory if subgoals with new definedness atoms are generated or if domain lemmas are applied. By this procedure, proof search for domain properties is extended.

If we extend proof search with generous markings, many additional proof steps may be non-contributing. Therefore, generous markings on their own often reduce the efficiency of proof search. But in combination with upward propagation (cf. Chapter 7), this weakness is compensated: Non-contributing proof steps which may contain open proof obligations are eliminated with hindsight.

Example 8.11 We analyze the effect of generous markings w.r.t. definedness atoms on the proof of Goal (8.208) in Example 8.10. With generous markings, the first proof attempt succeeds as sketched in Figure 8.18 since all literals remain mandatory in the first condition subgoal. Essentially, the proof consists of the first proof attempt of Example 8.10 as illustrated in Figure 8.15 and the proof of the wellformedness property depicted in Figure 8.16. Additionally, it contains four unnecessary proof steps.

Table 8.2 contains statistics for the 66 Lemmas about `Lpo`. It compares configurations with generous markings and reuse enabled (resp. disabled) as indicated with "✓" (resp. "—"):

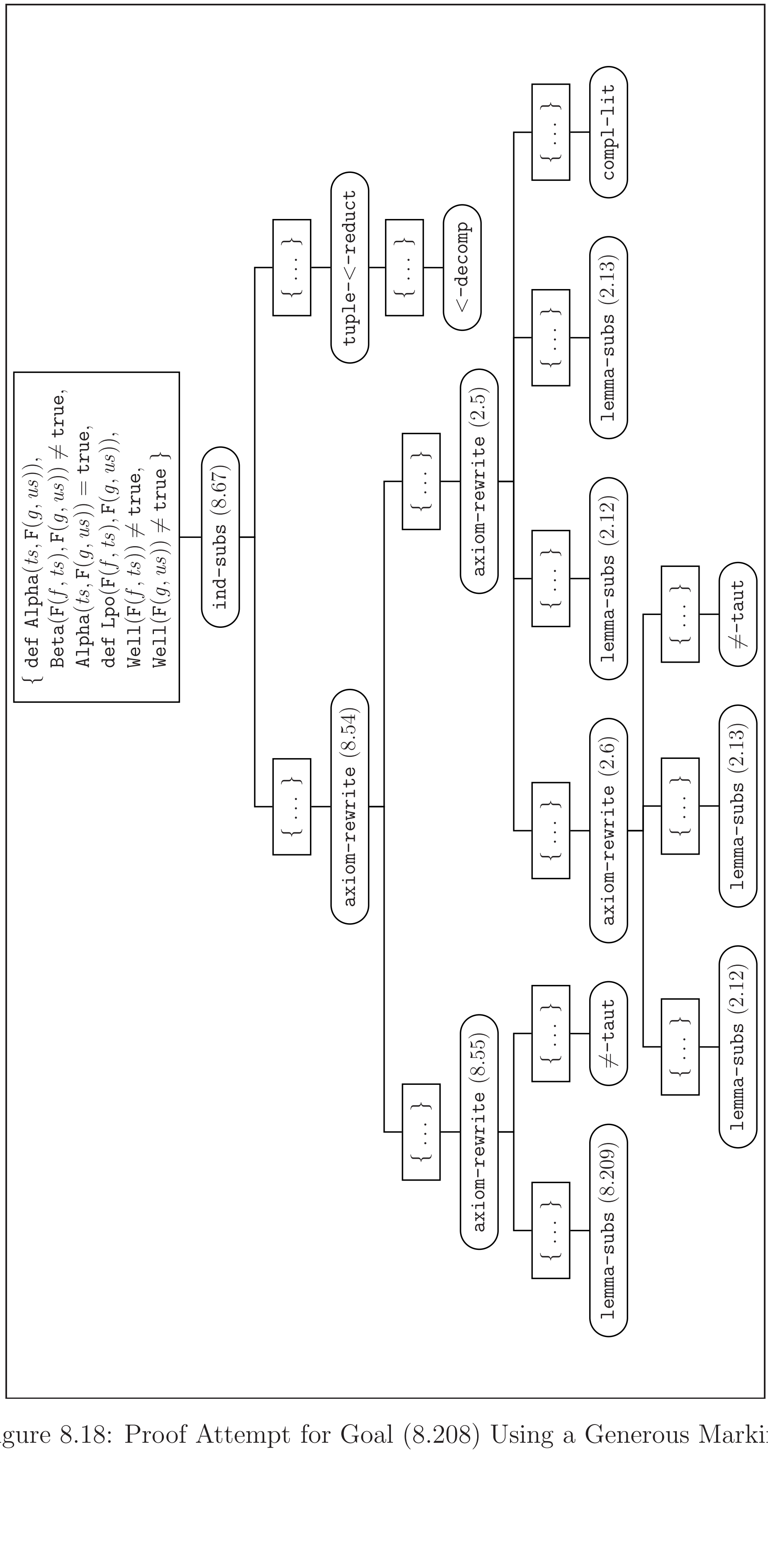

Figure 8.18: Proof Attempt for Goal (8.208) Using a Generous Marking

Generous Markings	Reuse	Autom. Appl.	Del.	Fin. P.	Runtime
—	—	4511	1152	3359	58.74
✓	—	5219	1381	3838	72.64
—	✓	4329	1028	3301	43.66
✓	✓	4462	1020	3442	39.29

Table 8.2: Statistics for the 66 Lemmas about `Lpo` w.r.t. Generous Markings and Reuse

- Comparing the statistics with reuse disabled, we notice that both—the runtime and the size of the final proofs—increase with generous markings enabled. This is caused by non-contributing proof steps which are introduced with generous markings and cannot be eliminated afterwards (since reuse is disabled).
- With reuse enabled, generous markings reduce the runtime because fewer proof attempts fail. Therefore, they improve the efficiency of proof search although the size of the final proofs increases.

This illustrates that the mechanisms introduced with generous markings and reuse benefit from each other: Generous markings relax proof search in such a way that fewer proof attempts fail. Our reuse mechanism—more precisely, upward propagation—enables proof control to eliminate unnecessary proof steps containing additional proof obligations with hindsight. The combination of both mechanisms improves the efficiency of our proof control. Nevertheless, the use of specialized auxiliary lemmas is still superior but at the expense of having to speculate these auxiliary lemmas, usually by user-interaction. □

In addition to the proofs of domain properties, we may use generous markings for the defining rules of "auxiliary" operators which are specified without recursion in terms of other "main" operators. If we mark every literal in the defining rules of an auxiliary operator as generous, terms starting with this auxiliary operator are reduced with the defining rules in any case. The resulting condition subgoals contain additional information about the main operators which may be exploited for the proofs of the condition subgoals. Therefore, we do not have to specify auxiliary lemmas for the auxiliary operators. We use this approach for the auxiliary operators `lpoR4` and `clpo4`.

Example 8.12 The following axioms define the auxiliary operator `lpoR4` in terms of syntactic equality and the main operator `lpo4`:

$$\{\ \mathtt{lpoR4}(t,u) = \mathtt{E},\ t \neq u\ \} \tag{8.213}$$

$$\{\ \mathtt{lpoR4}(t,u) = \mathtt{G},\ t = u,\ \mathtt{lpo4}(t,u) \neq \mathtt{true}\ \} \tag{8.214}$$

$$\{\ \mathtt{lpoR4}(t,u) = \mathtt{N},\ t = u,\ \mathtt{lpo4}(t,u) = \mathtt{true},\ \neg\mathtt{def}\ \mathtt{lpo4}(t,u)\ \} \tag{8.215}$$

For `lpoR4`, we do not define any auxiliary lemmas (except of its domain lemma). Instead, we mark all the literals in Axioms (8.213) to (8.215) as generous. Each term with top-level operator `lpoR4` may be reduced using these axioms. The mandatory marking of the parent goal is inherited to the generated condition subgoals. The added literals in the generated condition subgoals contain information about the main operator `lpo4` which may facilitate the proofs of the condition subgoals.

If we did not use these generous markings, four goals would remain open in the equivalence proof between variant `lpoR5` and `lpoR4`. Thus, we would have to speculate additional auxiliary lemmas to perform the proofs. □

8.3 Concluding Remarks

In this chapter, we have validated our new proof techniques with a comprehensive case study. The case study about the LPO is challenging because of its omnipresent use of mutual recursion which calls for proofs based on mutual induction. In QUODLIBET, the original definitions can be formalized in a very natural and straightforward way using partially defined operators. Proofs based on mutual induction cause difficulties in speculating auxiliary lemmas for all dependent operators, in performing appropriate inductive case splits, in applying suitable induction hypotheses, in finding suitable wellfounded induction orders, and in proving the corresponding order constraints. Our proof process based on descente infinie supports us in solving these tasks as the required information has to be supplied just when needed. Therefore, we can exploit intermediate results for acquiring the information. We may, for instance, delay the choice of the induction order by instantiating the weight variables until all order constraints are present.

We cannot imagine to perform such complicated proofs based on mutual induction completely automatically. Therefore, our proof techniques aim at supporting manual interactions in a suitable way. In this chapter, we have described the process itself as well as the proof script resulting from our proof engineering process.

In the case study about the LPO, we apply all of our new proof techniques: the integration of Hodes' decision procedure for linear arithmetic; guiding proof search with mandatory, obligatory and generous markings; and the new reuse mechanisms consisting of upward propagation and sideward reuse.

At the moment, Hodes' decision procedure is used only rudimentally for simplifying the proofs of order constraints that exploit the commutativity of addition (cf. Example 8.7). In the future, we intend to perform all the proofs of order constraints with the decision procedure. We will concretize this approach in Section 8.3.1.

Conditional lemmas are omnipresent in inductive theorem proving. Thus, the speculation and automatic application of auxiliary lemmas is essential for the success of the proof process. Auxiliary lemmas may be derived from open goals of failed proof attempts by generalization. Thus, our proof control supports the user in this task. Nevertheless, we do not perform it fully automatically because, for complicated lemmas, the generalization cannot be based on purely syntactical considerations: It requires some kind of ingenuity based on domain knowledge. This manifests itself in the need for new operators to specify auxiliary lemmas as e.g. in Example 8.9.

We have developed new proof techniques to control the automatic application of conditional lemmas using markings in goals and lemmas. The usefulness of mandatory and obligatory markings are demonstrated in Chapter 6. In this chapter, we have focused on the benefits of generous markings which allow us to abandon some auxiliary lemmas. Altogether, our approach based on markings provides for a fine-grained manual control of proof search which allows the user to change the default behavior in various ways. This is particularly useful for supporting the interactions required for complicated proofs in inductive theorem proving.

Our reuse mechanism is based on the contribution of proof steps in a performed proof. Upward propagation simplifies proofs by eliminating non-contributing proof steps. In doing this, open proof obligations may be eliminated as well. But according to Table 7.2 in Section 7.4, upward propagation is rarely applicable within our proof control. This is caused by the fact that our proof control favors inference rules that use new literals and thus result in contributing proof steps. Additionally, our proof control prevents most of the non-contributing proof steps due to the use of mandatory markings. In the case study about the LPO, the mandatory markings heuristics are relaxed by using generous markings. Example 8.11 demonstrates the synergetic effect which results from the combination of generous markings and upward propagation.

8.3.1 Future Improvements

The case study about the LPO allowed us to validate new proof techniques such as mandatory and obligatory markings for guiding proof search. Furthermore, it initiated the development of new proof techniques such as generous markings. In this section, we want to sketch further insights in the development of new proof techniques which have not been realized so far.

The integration of Hodes' decision procedure for linear arithmetic calls for a stronger provision of this procedure for proving order constraints in inductive proofs. This may be realized by representing the present wellfounded order based on the length of constructor ground terms explicitly. For this, we may define an operator s-$\mathtt{size}$ for each sort s automatically. More precisely, if s is defined with constructors $\mathtt{c}_i : s_{i,1}, \ldots, s_{i,k_i} \rightarrow s_i$ for $i \in \{1, \ldots, n\}$, then s-$\mathtt{size}$ may be specified with the following n defining rules

$$s\text{-}\mathtt{size}(\mathtt{c}_i(x_{i,1}, \ldots, x_{i,k_i})) = 1 + \sum_{j=1}^{k_i} s_{i,j}\text{-}\mathtt{size}(x_{i,j}) \quad \text{for } i \in \{1, \ldots, n\}.$$

This allows us to perform the proofs of order constraints using the decision procedure instead of the inference rules $<$-$\mathtt{decomp}$, $<$-$\mathtt{mono}$, and $<$-$\mathtt{trans}$. In doing this, we hope to improve the degree of automation and the efficiency in proving order constraints.

Examples 8.6 and 8.7 provide evidence that, in some cases, it would be beneficial to use a multiset extension for comparing order constraints instead of the given lexicographic order. Unfortunately, we have to perform all comparisons of order constraints in inductive proofs with *one* fixed wellfounded order. But this restriction can be softened by using defined function symbols during the instantiation of the weight variables. Since single components are compared w.r.t. the length of the corresponding constructor terms using the usual *total* order on natural numbers, it is possible to simulate a multiset extension in an easy way. In fact, the comparison of finite tuples w.r.t. a multiset extension of a total order can be achieved by

- sorting the tuples on both sides in decreasing order; and
- comparing the sorted tuples with a lexicographic order.

Therefore, it is possible to combine a multiset extension with a lexicographic order. In principle, we require only two defined operators for sorting tuples of natural numbers and for projecting single components from a tuple of natural numbers. But using these defined operators would by far be too inefficient.

Instead, we suggest to adapt the approach used for the integration of Hodes' decision procedure: We suggest to extend the inference rules `<-taut`, `<-removal`, `tuple-<-reduct`, and `tuple-=-reduct`, which realize the lexicographic comparison, in such a way that they may also be used for the multiset extension. Weight variables may be instantiated with "unsorted" tuples of terms as before or with "decreasingly sorted" tuples of terms. The difference may be denoted during the instantiation of the weight variable by using a special keyword such as `sort` in the latter case. An order constraint consisting of two unsorted tuples is compared with the lexicographic order as before. An order constraint consisting of two sorted tuples is compared with the multiset extension. In fact, we may even compare a sorted tuple with an unsorted tuple if the order constraint results from a proof based on mutual induction. The provision of specialized inference rules will improve the efficiency of the approach: One application of the new inference rules will replace many rewrite steps with the axioms of the defined operators for sorting and projecting.

Since finding a suitable induction order and proving the order constraints with this induction order is one of the most important and challenging problems in our case study, it may be beneficial to consider further proof techniques for automating termination proofs, such as the *dependency pair* approach [AG00, TG05]. In the future, we may integrate these techniques into QuodLibet or use a specialized external proof system, such as `AProVE` [GTSKF04], for performing the termination proofs.

Chapter 9

Conclusion

In this thesis, we have developed techniques for the automation of theorem proving with a special focus on user-interaction. In our conclusion, we want to discuss our work in a wider context.

The necessity for user-interaction. In theorem proving, the search space is usually so huge that domain knowledge is needed for guiding proof search into the right direction. In inductive theorem proving, we have to cope with additional problems: From a theoretical point of view, the inductive validity is not even semi-decidable. In practice, additional lemmas have to be speculated or generalized which may even have to contain new operators. For well known domains, we may integrate the needed domain knowledge directly into the proof process. But in general, user-interaction is indispensable for speculating auxiliary lemmas and guiding proof search.

The state-of-the-art. Nowadays, it is much easier to "sell" improvements w.r.t. the automation of a theorem prover than w.r.t. the integration of user-interaction. Probably this is caused by the fact that the latter improvements can hardly be measured whereas the former ones are easily shown by decreased runtimes or fewer applications of inference rules. Therefore, user-interaction is handled in many systems only rudimentally. Nevertheless, the importance of a knowledgeable user to guide the proof process is recognized and described e.g. for `ACL2` in [KMM00]. In `ACL2`, the user mainly guides proof search by speculating and activating suitable auxiliary lemmas. Furthermore, the user may provide *hints*. With hints, phases of the waterfall may be excluded from the proof process, disabling, for instance, the cross-fertilization or the induction phase; induction schemes may be provided by terminating recursion schemes of defined operators; or the application of special lemma instances may be forced. To our knowledge, there is no way to influence proof search in `ACL2` on a more fine-grained level. The relief test for conditional lemmas, for instance, is fixed w.r.t. Contextual Rewriting.

Additional demands on interactive theorem proving. The need for user-interaction poses additional demands on the whole proof system and, in particular, on the techniques for automating proofs. On the basis of our experience with complex case studies such as the one about the lexicographic path order LPO, we recommend that such an interactive proof system (resp. its automatic proof control) should be

flexible: The user has to be able to interact with the proof system on various levels. The needed information should be acquired in a suitable way when needed.

restricted: Automated proof attempts should stop within a reasonable amount of time allowing for the analysis of failed proof attempts.

informative: The system should support the user in analyzing (failed) proof attempts. For this, the information provided by the system should be

  understandable: The output should be readable for humans in an easy way.

  comprehensive: The output should provide the user with as much useful information as possible.

  essential: The output should provide the user with as little unnecessary information as possible.

Our contributions. For the most part, our new proof techniques are independent from a special proof system. Nevertheless, we have integrated them into the inductive theorem prover QuodLibet to validate their applicability. From the outset, QuodLibet has been developed with a special focus on user-interaction. Therefore, it offers special features that support this task as e.g. an inductive proof process based on descente infinie. In the following, we discuss the benefits of our new proof techniques w.r.t. the criteria established for interactive theorem proving above:

- Our close integration of Hodes' decision procedure for linear arithmetic provides the user with information that is hidden in the internal state of the decision procedure in other approaches. Furthermore, it allows the user to derive further useful consequences automatically. This supports the user in speculating auxiliary lemmas needed for patching failed proof attempts. In our approach, the output is better *understandable* and more *comprehensive*.

- Our new framework for controlling proof search based on markings is very *flexible*. It enables to user to combine different approaches on an abstract level. The extent and efficiency of proof search can be controlled on a fine-grained level, i.e. for each lemma separately. This allows the user to *restrict* proof search in a suitable way.

- Our new reuse techniques supplement our proof search techniques. The proposed reuse techniques enable the derivation of minimized proofs based on the notion of contribution. In doing so, unnecessary information is eliminated automatically. Therefore, the user may focus on the *essential* information.

We have demonstrated the usefulness of these proof techniques by performing complex case studies with QuodLibet such as the case study about the LPO. This case study is challenging in particular because of the heavy use of mutual recursion/induction.

Future directions of research. Naturally, we could not "solve" the problem of integrating user-interaction into (inductive) theorem provers. Nevertheless, we hope that our work will motivate other researchers to spend more time for developing proof techniques that support user-interaction in a suitable way.

In addition to this, there is always the need for better automation. In the previous chapters we have already pointed out some topics for future developments:

- The integration of linear arithmetic (cf. Chapter 4) may be extended to non-linear arithmetic. The close integration of other theories and their combination may be considered as well.
- The effects of the different markings controlling proof search (cf. Chapter 6), in particular the effects resulting from their combination, call for a more detailed analysis.
- Enhancing the applicability of upward propagation (cf. Chapter 7) would be beneficial to improve reusability.
- The proof engineering process sketched in Chapter 8 may be used for deriving further proof techniques, e.g. for computing suitable induction orders.

Therefore, much promising work still is to be done.

Appendix A

The Proof Script for the Case Study About LPO

In this appendix, we present the complete final version of our proof script for the case study about the lexicographic path order LPO. It can be executed with the final version of the core system of QuodLibet and the proof control developed in this thesis. The final version contains Hodes' decision procedure for linear arithmetic (cf. Chapter 4); proof search is controlled with mandatory, obligatory, and generous literals (cf. Chapter 6); and the reuse mechanisms (cf. Chapter 7) are enabled.

With this appendix, we want to illustrate the size and the complexity of the case study, and the readability of the resulting proof scripts. We neither discuss the proof engineering process nor comment on the resulting proof script in detail. For parts of the specification, this is done in Chapter 8. Here, we just give a short overview of the command language used for the proof script. Further details may be found in [Küh00, Kai02, SS04].

For our proof script, the following commands are relevant:

`initialize` resets the internal data structures of the inference machine kernel.

`execute` loads a proof script from a file.

`define sort` introduces a new sort together with its constructors.

`declare constructor variables` introduces a set of new constructor variables.

`declare operators` introduces a set of new defined operators fixing their signatures.

`assert` allows the user to specify the defining rules of defined operators. The defining rules are given by conditional equations.

`assume` introduces a new conjecture. For each conjecture, a new proof state tree is generated which is used for representing inductive proof attempts of the conjecture.

`set weight` allows the user to instantiate the weight variable of a proof state tree.

`apply` allows the user to apply an inference rule manually to a goal node in a proof state tree.

`call` executes a public QML-routine. Our automatic proof control provides

- the following procedures:

  `initialize-database` resets the internal data structures of our proof control.

  `analyze-operator` performs a recursion analysis of the defining rules of a defined operator. By default, the defining rules are activated, i.e. they are checked for applicability during the simplification process. Furthermore, a domain lemma for the operator is generated if possible. The default behavior may be changed with optional parameters.

  `activate-axiom(s)/(-linear)-lemma(s)` activates (a set of) defining rules or lemmas in such a way that their applicability is checked during the simplification process.

  `deactivate-axioms/-lemmas` changes the state of the proof control in such a way that the given axioms or lemmas are not considered for automatic applications during the simplification process anymore.

  `set-default-settings` changes the default behavior of the procedures and tactics of the proof control globally.

- the following tactics which are applied to a goal node in a proof state tree:

  `auto-/operators-/variables-strategy` starts an automatic proof attempt for the root goal of a proof state tree. The initial case split is generated automatically (`auto`), semi-automatically (`operators`) or manually (`variables`), cf. Section 3.1.4 for more details.

  `cont-proof-attempt` continues a proof attempt by simplifying each open goal in the proof state tree.

  `simplify` applies the simplification process to a goal.

A proof script is parsed line by line. If at the end of a line a complete command has been parsed, this command will be executed. Otherwise, parsing is continued in the next line. The same behavior may be forced by ending a line with two dots. This allows the user to split a command into two lines even if the first line contains a complete command. Lines starting with two slashes are considered as comments and ignored. The behavior of QML-routines may be changed with optional parameters. The values of these parameters are preceded with keywords which start with a colon and identify the corresponding optional parameters.

A.1 The Overall Proof Script: `Lpo-all`

The proof script of the case study is partitioned into different files: one file for each sort, one file for each variant of the LPO, and one file for the whole proof. This file essentially initializes the core system and the proof control and executes the remaining files.

```
initialize
call initialize-database
call set-default-settings :allow-alternative-free-var-bindings-p FALSE
```

```
execute "basics" echo
execute "bool" echo
execute "res" echo
execute "nat" echo
execute "term" echo
execute "Lpo" echo
execute "lpo1" echo
execute "lpo2" echo
execute "lpo3" echo
execute "lpo4" echo
execute "lpoR4" echo
execute "lpoR5" echo
execute "lpoR6" echo
execute "clpo4" echo
execute "clpo5" echo
execute "clpo6" echo
```

A.2 basics

This file contains the defining rules of the predefined operators resulting from the integration of linear arithmetic (cf. Chapter 4).

```
declare constructor variables cv-Nat_1, cv-Nat_2, cv-Nat_3 : Nat.

assert
  +-1 :
    +(cv-Nat_1,0) = cv-Nat_1
  +-2 :
    +(cv-Nat_1,s(cv-Nat_2)) = s(+(cv-Nat_1,cv-Nat_2))
    .
call analyze-operator  + :auto-insert-axioms-p FALSE
// analyze-operator generates the domain lemma
// { def +(cv-Nat_1,cv-Nat_2) } named +-def-auto
call auto-strategy  +-def-auto
call activate-lemma  +-def-auto

assert
  *-1 :
    *(cv-Nat_1,0)=0
  *-2 :
    *(cv-Nat_1,s(cv-Nat_2))=+(*(cv-Nat_1,cv-Nat_2),cv-Nat_1)
    .
call analyze-operator * :auto-insert-axioms-p FALSE
// analyze-operator generates the domain lemma
// { def *(cv-Nat_1,cv-Nat_2) } named *-def-auto
call auto-strategy  *-def-auto
call activate-lemma  *-def-auto

assert
  --1 :
    -(cv-Nat_1,0) = cv-Nat_1
  --2 :
    -(0,s(cv-Nat_2)) = 0
  --3 :
```

```
    -(s(cv-Nat_1),s(cv-Nat_2)) = -(cv-Nat_1,cv-Nat_2)
    .
call analyze-operator  - :auto-insert-axioms-p FALSE
// analyze-operator generates the domain lemma
// { def -(cv-Nat_1,cv-Nat_2) } named --def-auto
call auto-strategy  --def-auto
call activate-lemma  --def-auto
```

A.3 Bool

This file defines a sort for boolean values and contains basic properties of conjunctions and disjunctions.

```
define sort Bool with constructors
    true :  --> Bool
    false :  --> Bool
    .
declare constructor variables
    b, b1, b2, b3, b4 : Bool.

assume
    { b = true,
      b = false }
    bool-complete
call auto-strategy  bool-complete
call activate-lemma  bool-complete

declare operators
    and : Bool Bool --> Bool
    .
assert
  and-1 :
    and(true,b2) = b2
  and-2 :
    and(false,b2) = false
    .
call analyze-operator  and
// analyze-operator generates the domain lemma
// { def and(b2,b) } named and-def-auto
call auto-strategy  and-def-auto
call activate-lemma  and-def-auto

assume
    { and(b1,b2) = and(b2,b1) }
    and-commutative
call auto-strategy :recursive-strategy-p TRUE  and-commutative
call activate-lemma  and-commutative

assume
    { and(b1,and(b2,b3)) = and(b2,and(b1,b3)) }
    and-extended-commutativity
call auto-strategy  and-extended-commutativity
call activate-lemma  and-extended-commutativity
```

```
assume
    { and(and(b1,b2),b3) = and(b1,and(b2,b3)) }
    and-associative
call auto-strategy  and-associative
call activate-lemma  and-associative

assume
    { and(b1,b2) = false,
      b1 = true }
    and-true-1
call auto-strategy  and-true-1
call activate-lemma  and-true-1

assume
    { and(b1,b2) = false,
      b2 = true }
    and-true-2
call auto-strategy  and-true-2
call activate-lemma  and-true-2

assume
    { and(b1,b2) = b2,
      b1 = false }
    and-false-1
call auto-strategy  and-false-1
call activate-lemma  and-false-1

assume
    { and(b1,b2) = b1,
      b2 = false }
    and-false-2
call auto-strategy  and-false-2
call activate-lemma  and-false-2

declare operators
    or : Bool Bool --> Bool
    .
assert
  or-1 :
    or(true,b2) = true
  or-2 :
    or(false,b2) = b2
    .
call analyze-operator  or
// analyze-operator generates the domain lemma
// { def or(b2,b) } named or-def-auto
call auto-strategy  or-def-auto
call activate-lemma  or-def-auto

assume
    { or(b1,b2) = or(b2,b1) }
    or-commutative
call auto-strategy :recursive-strategy-p TRUE  or-commutative
call activate-lemma  or-commutative

assume
```

```
    { or(b1,or(b2,b3)) = or(b2,or(b1,b3)) }
    or-extended-commutativity
call auto-strategy  or-extended-commutativity
call activate-lemma  or-extended-commutativity

assume
    { or(or(b1,b2),b3) = or(b1,or(b2,b3)) }
    or-associative
call auto-strategy  or-associative
call activate-lemma  or-associative

assume
    { or(b1,b2) = b2,
      b1 = true }
    or-true-1
call auto-strategy  or-true-1
call activate-lemma  or-true-1

assume
    { or(b1,b2) = b1,
      b2 = true }
    or-true-2
call auto-strategy  or-true-2
call activate-lemma  or-true-2

assume
    { or(b1,b2) = true,
      b1 = false }
    or-false-1
call auto-strategy  or-false-1
call activate-lemma  or-false-1

assume
    { or(b1,b2) = true,
      b2 = false }
    or-false-2
call auto-strategy  or-false-2
call activate-lemma  or-false-2
```

A.4 Res

This file defines sort `Res` required for the variants `lpoR` (cf. Sections A.12 to A.14) and `clpo` (cf. Sections A.15 to A.17).

```
define sort Res with constructors
    E :  --> Res
    G :  --> Res
    L :  --> Res
    N :  --> Res
    .
declare constructor variables
    r, r1, r2, r3, r4 : Res.

assume
```

```
    { r = E,
      r = G,
      r = L,
      r = N }
    res-complete
call auto-strategy  res-complete
call activate-lemma  res-complete

declare operators
    flip : Res --> Res
    .
assert
  flip-1 :
    flip(E) = E
  flip-2 :
    flip(N) = N
  flip-3 :
    flip(L) = G
  flip-4 :
    flip(G) = L
    .
call analyze-operator  flip
// analyze-operator generates the domain lemma
// { def flip(r) } named flip-def-auto
call auto-strategy  flip-def-auto
call activate-lemma  flip-def-auto
```

A.5 Nat

Since the predefined operators of sort `Nat` are already defined in file `basics` (cf. Section A.2), in this file we just introduce some additional constructor variables.

```
declare constructor variables
    n, m, n1, n2, n3, n4, m1, m2, m3, m4 : Nat
    .
```

A.6 term

This file contains the definitions and lemmas for terms and lists of terms which mutually depend on each other. Most important are the wellformedness properties and different types of subterm relations.

```
define sort FID with constructors
    Fid : Nat Nat --> FID
    .
declare constructor variables
    f, g, h, f1, f2, f3, f4 : FID
    .

declare operators
```

```
    prec : FID FID --> Bool
    .
assert
  prec-1:
    prec(Fid(n,m),Fid(n1,m1)) = true
      if ~(n <= n1)
  prec-2:
    prec(Fid(n,m),Fid(n1,m1)) = false
      if n <= n1
    .
call analyze-operator  prec
// analyze-operator generates the domain lemma
// { def prec(f,g) } named prec-def-auto
call auto-strategy  prec-def-auto
call activate-lemma  prec-def-auto

assume
    { prec(f,g) = true,
      prec(f,h) =/= true,
      prec(h,g) =/= true }
    prec-trans
call auto-strategy  prec-trans
call activate-lemma  prec-trans

assume
    { prec(f,f) = false }
    prec-irrefl
call auto-strategy  prec-irrefl
call activate-lemma  prec-irrefl

assume
    { prec(f,g) =/= true,
      prec(g,f) =/= true }
    prec-irrefl-trans
apply lemma-subs
    prec-trans
    [f <-- f , h <-- g, g <-- f] ..
    prec-irrefl-trans
call simplify  prec-irrefl-trans
call activate-lemma  prec-irrefl-trans :head-litnbs { 1 } :obl-litnbs-list { { 2 } }

declare operators
    arity : FID --> Nat
    .
assert
  arity-1 :
    arity(Fid(n,m)) = m
    .
call analyze-operator  arity
// analyze-operator generates the domain lemma
// { def arity(f) } named arity-def-auto
call auto-strategy  arity-def-auto
call activate-lemma  arity-def-auto

define sort VID with constructors
    Vid : Nat --> VID
```

```
    .
declare constructor variables
    x, y, z, x1, x2, x3, x4 : VID
    .

declare sorts Term.
declare sorts Termlist.
define sort Term with constructors
    V : VID --> Term
    F : FID Termlist --> Term
    .
define sort Termlist with constructors
    nil :  --> Termlist
    cons : Term Termlist --> Termlist
    .
declare constructor variables
    ts, us, vs, ws, ts1, ts2, ts3, ts4, us1 : Termlist.
declare constructor variables
    t, u, v, w, t1, t2, t3, t4, ti, ui : Term.

declare operators
    length : Termlist --> Nat
    .
assert
  length-1 :
    length(nil) = 0
  length-2 :
    length(cons(u,us)) = s(length(us))
    .
call analyze-operator  length
// analyze-operator generates the domain lemma
// { def length(us) } named length-def-auto
call auto-strategy  length-def-auto
call activate-lemma  length-def-auto

declare operators
    Fun : Term --> Bool
    .
assert
  Fun-1 :
    Fun(F(f,ts)) = true
  Fun-2 :
    Fun(V(x)) = false
    .
call analyze-operator Fun
// analyze-operator generates the domain lemma
// { def Fun(t) } named Fun-def-auto
call auto-strategy Fun-def-auto
call activate-lemma Fun-def-auto

declare operators
    Var : Term --> Bool
    .
assert
  Var-1 :
    Var(V(x)) = true
```

```
  Var-2 :
    Var(F(f,ts)) = false
    .
call analyze-operator Var
// analyze-operator generates the domain lemma
// { def Var(t) } named Var-def-auto
call auto-strategy Var-def-auto
call activate-lemma Var-def-auto

declare operators
    Well : Term --> Bool
    Well_tl : Termlist --> Bool
    .
assert
  Well-1 :
    Well(V(x)) = true
  Well-2 :
    Well(F(f,ts)) = Well_tl(ts)
      if arity(f) = length(ts),
         def arity(f),
         def length(ts)
  Well-3 :
    Well(F(f,ts)) = false
      if arity(f) =/= length(ts),
         def arity(f),
         def length(ts)
  Well_tl-1 :
    Well_tl(nil) = true
  Well_tl-2 :
    Well_tl(cons(u,us)) = Well_tl(us)
      if Well(u) = true
  Well_tl-3 :
    Well_tl(cons(u,us)) = false
      if Well(u) =/= true, def Well(u)
    .
call analyze-operator  Well
// analyze-operator generates the domain lemma
// { def Well(t) } named Well-def-auto
call analyze-operator  Well_tl
// analyze-operator generates the domain lemma
// { def Well_tl(us) } named Well_tl-def-auto
set weight t Well-def-auto
set weight us Well_tl-def-auto
call auto-strategy :ind-lemmas { Well_tl-def-auto Well-def-auto } Well-def-auto
call auto-strategy :ind-lemmas { Well-def-auto Well_tl-def-auto } Well_tl-def-auto
call activate-lemmas { Well_tl-def-auto Well-def-auto }

assume
    { Well_tl(cons(t,ts)) = false,
      Well_tl(ts) = true }
    Well_tl-Well_tl
call simplify  Well_tl-Well_tl
call activate-lemma  Well_tl-Well_tl :obl-litnbs-list { { 2 } }

assume
    { Well(F(f,ts)) = false,
```

```
      Well_tl(ts) = true }
    Well_tl-Well
call simplify Well_tl-Well
call activate-lemma  Well_tl-Well :obl-litnbs-list { { 2 } }

assume
    { Well(F(f,us)) =/= true,
      Well(F(f,ts)) =/= true,
      length(us) = length(ts) }
    length-Well
call simplify length-Well
call activate-lemma  length-Well :head-litnbs { 1 } :obl-litnbs-list { { 2 3 } } ..
                                :free-vars-bindings { { "lit(3)" } }

declare operators
    contains_tl : Termlist VID --> Bool
    contains : Term VID --> Bool
    .
assert
  contains_tl-1 :
    contains_tl(nil,y) = false
  contains_tl-2 :
    contains_tl(cons(t,ts),y) = true
        if contains(t,y) = true
  contains_tl-3 :
    contains_tl(cons(t,ts),y) = contains_tl(ts,y)
      if contains(t,y) =/= true,
         def contains(t,y)
  contains-1 :
    contains(V(x),y) = true
      if x = y
  contains-2 :
    contains(V(x),y) = false
      if x =/= y
  contains-3 :
    contains(F(f,ts),y) = contains_tl(ts,y)
    .
call analyze-operator  contains
// analyze-operator generates the domain lemma
// { def contains(t,y) } named contains-def-auto
call analyze-operator  contains_tl
// analyze-operator generates the domain lemma
// { def contains_tl(ts,y) } named contains_tl-def-auto
set weight ts contains_tl-def-auto
set weight t contains-def-auto
call auto-strategy :ind-lemmas { contains_tl-def-auto contains-def-auto } ..
                   contains-def-auto
call auto-strategy :ind-lemmas { contains-def-auto contains_tl-def-auto } ..
                   contains_tl-def-auto
call activate-lemmas { contains_tl-def-auto contains-def-auto }

declare operators
    subterm : Term Term --> Bool
    subterm_tl : Term Termlist --> Bool
    .
assert
```

```
  subterm-1 :
    subterm(t,V(y)) = false
  subterm-2 :
    subterm(t,F(g,us)) = subterm_tl(t,us)
  subterm_tl-1 :
    subterm_tl(t,nil) = false
  subterm_tl-2 :
    subterm_tl(t,cons(u,us)) = true
      if t = u
  subterm_tl-3 :
    subterm_tl(t,cons(u,us)) = true
      if t =/= u,
         subterm(t,u) = true
  subterm_tl-4 :
    subterm_tl(t,cons(u,us)) = subterm_tl(t,us)
      if t =/= u,
         subterm(t,u) =/= true,
         def subterm(t,u)
    .
call analyze-operator  subterm
// analyze-operator generates the domain lemma
// { def subterm(t,u) } named subterm-def-auto
call analyze-operator  subterm_tl
// analyze-operator generates the domain lemma
// { def subterm_tl(t,us) } named subterm_tl-def-auto
set weight u subterm-def-auto
set weight us subterm_tl-def-auto
call auto-strategy :ind-lemmas { subterm_tl-def-auto subterm-def-auto } ..
                   subterm-def-auto
call auto-strategy :ind-lemmas { subterm-def-auto subterm_tl-def-auto } ..
                   subterm_tl-def-auto
call activate-lemmas { subterm_tl-def-auto subterm-def-auto }

assume
    { Well(u) = true,
      Well(t) =/= true,
      subterm(u,t) =/= true}
    subterm-Well
assume
    { Well(u) = true,
      Well_tl(ts) =/= true,
      subterm_tl(u,ts) =/= true}
    subterm-Well_tl
set weight t subterm-Well
set weight ts subterm-Well_tl
call auto-strategy :ind-lemmas { subterm-Well subterm-Well_tl} subterm-Well
call auto-strategy :ind-lemmas { subterm-Well subterm-Well_tl} subterm-Well_tl
call activate-lemma subterm-Well :obl-litnbs-list { { 2 } }
call activate-lemma subterm-Well_tl :obl-litnbs-list { { 2 } }

assume
    { subterm(t,u) = true,
      subterm(t,v) =/= true,
      subterm(v,u) =/= true }
    subterm-trans
call auto-strategy :recursive-strategy-p TRUE  subterm-trans
```

```
call activate-lemma  subterm-trans

assume
    { subterm_tl(u,cons(t,ts)) = true,
      subterm_tl(u,ts) =/= true }
    subterm_tl-cons
call simplify subterm_tl-cons
call activate-lemma  subterm_tl-cons


declare operators
    subterms : Termlist Term --> Bool
    .
assert
  subterms-1 :
    subterms(nil,u) = true
  subterms-2 :
    subterms(cons(t,ts),u) = subterms(ts,u)
      if subterm(t,u) = true
  subterms-3 :
    subterms(cons(t,ts),u) = false
      if subterm(t,u) =/= true,
         def subterm(t,u)
    .
call analyze-operator  subterms
// analyze-operator generates the domain lemma
// { def subterms(ts,t) } named subterms-def-auto
call auto-strategy  subterms-def-auto
call activate-lemma  subterms-def-auto

declare operators
    subterms_tl : Termlist Termlist --> Bool
    .
assert
  subterms_tl-1 :
    subterms_tl(nil,us) = true
  subterms_tl-2 :
    subterms_tl(cons(t,ts),us) = subterms_tl(ts,us)
      if subterm_tl(t,us) = true
  subterms_tl-3 :
    subterms_tl(cons(t,ts),us) = false
      if subterm_tl(t,us) =/= true,
         def subterm_tl(t,us)
    .
call analyze-operator  subterms_tl
// analyze-operator generates the domain lemma
// { def subterms_tl(ts,us) } named subterms_tl-def-auto
call auto-strategy  subterms_tl-def-auto
call activate-lemma  subterms_tl-def-auto

assume
    { subterms_tl(us,cons(t,ts)) = true,
      subterms_tl(us,ts) =/= true }
    subterms_tl-cons
call auto-strategy  subterms_tl-cons
call activate-lemma  subterms_tl-cons
```

```
assume
    { subterms(ts,F(g,us)) = true,
      subterms_tl(ts,us) =/= true }
    subterms-subterms_tl
call auto-strategy  subterms-subterms_tl
call activate-lemma  subterms-subterms_tl

assume
    { subterms_tl(ts,ts) = true }
    subterms_tl-refl
call auto-strategy  subterms_tl-refl
call activate-lemma  subterms_tl-refl

assume
    { subterm_tl(t,ts) = false,
      subterms(ts,t) =/= true }
    subterm_tl-subterms
assume
    { subterm(t,t) = false }
    subterm-irrefl
set weight ts subterm_tl-subterms
set weight t subterm-irrefl
call auto-strategy :ind-lemmas { subterm-irrefl subterm_tl-subterms } ..
                   subterm_tl-subterms
apply lemma-subs
    subterm-trans
    [t <-- u , v <-- t] ..
    subterm_tl-subterms
call simplify :ind-lemmas { subterm-irrefl } subterm_tl-subterms
call activate-ind-lemma  subterm_tl-subterms :obl-litnbs-list {  }
call auto-strategy :ind-lemmas { subterm_tl-subterms } subterm-irrefl
call activate-lemma  subterm-irrefl

declare operators
    sublist : Termlist Termlist --> Bool
    .
assert
  sublist-1 :
    sublist(nil,us) = true
  sublist-2 :
    sublist(cons(t,ts),nil) = false
  sublist-3 :
    sublist(cons(t,ts),cons(u,us)) = sublist(ts,us)
      if t = u
  sublist-4 :
    sublist(cons(t,ts),cons(u,us)) = sublist(cons(t,ts),us)
      if t =/= u
    .
call analyze-operator  sublist
// analyze-operator generates the domain lemma
// { def sublist(ts,us) } named sublist-def-auto
call auto-strategy  sublist-def-auto
call activate-lemma  sublist-def-auto

assume
```

```
    { sublist(ts,ts) = true }
    sublist-refl
call auto-strategy  sublist-refl
call activate-lemma  sublist-refl

assume
    { sublist(cons(u,ts),us) = false,
      sublist(ts,us) =/= false }
    sublist-cons
call auto-strategy  sublist-cons
call activate-lemma  sublist-cons

assume
    { sublist(cons(u,ts),us) = false,
      subterm_tl(u,us) = true }
    subterm_tl-sublist
call auto-strategy  subterm_tl-sublist
call activate-lemma  subterm_tl-sublist :obl-litnbs-list { { 2 } }

assume
    { sublist(cons(t,ts),us) = false,
      subterms(us,t) =/= true }
    sublist-subterms
call auto-strategy  sublist-subterms
call activate-lemma  sublist-subterms

assume
    { contains(u,y) = true,
      subterm(V(y),u) =/= true }
    contains-subterm
assume
    { contains_tl(ts,y) = true,
       subterm_tl(V(y),ts) =/= true }
    contains-subterm_tl
set weight u contains-subterm
set weight ts contains-subterm_tl
call auto-strategy :ind-lemmas { contains-subterm_tl contains-subterm } ..
                   contains-subterm
call auto-strategy :ind-lemmas { contains-subterm contains-subterm_tl } ..
                   contains-subterm_tl
call activate-lemmas { contains-subterm contains-subterm_tl }

declare operators
    term-arg_tl : Termlist Nat --> Term
    .
assert
  term-arg_tl-1 :
    term-arg_tl(cons(t,ts),s(0)) = t
  term-arg_tl-2 :
    term-arg_tl(cons(t,ts),s(s(n))) = term-arg_tl(ts,s(n))
    .
call analyze-operator  term-arg_tl
assume
    { def term-arg_tl(ts,n),
      n = 0,
      +(1,length(ts)) <= n }
```

```
    def-term-arg_tl
call auto-strategy  def-term-arg_tl
call activate-lemma  def-term-arg_tl

define sort Position with constructors
    nnil :  --> Position
    ncons : Nat Position --> Position
    .
declare constructor variables l : Position.

declare operators
    pos-p : Position Term --> Bool
    .
assert
  pos-p-1 :
    pos-p(nnil,t) = true
  pos-p-2 :
    pos-p(ncons(n,l),V(x)) = false
  pos-p-3 :
    pos-p(ncons(n,l),F(f,ts)) = false
      if n = 0
  pos-p-4 :
    pos-p(ncons(n,l),F(f,ts)) = false
      if n =/= 0,
         ~(n <= length(ts))
  pos-p-5 :
    pos-p(ncons(n,l),F(f,ts)) = pos-p(l,term-arg_tl(ts,n))
      if n =/= 0,
         n <= length(ts)
    .
call analyze-operator  pos-p
// analyze-operator generates the domain lemma
// { def pos-p(l,t) } named pos-p-def-auto
call auto-strategy  pos-p-def-auto
call activate-lemma  pos-p-def-auto

declare operators
    elem : Term Termlist --> Bool
    .
assert
  elem-1 :
    elem(t,nil) = false
  elem-2 :
    elem(t,cons(u,us)) = true
      if t = u
  elem-3 :
    elem(t,cons(u,us)) = elem(t,us)
      if t =/= u
    .
call analyze-operator  elem
// analyze-operator generates the domain lemma
// { def elem(t,us) } named elem-def-auto
call auto-strategy  elem-def-auto
call activate-lemma  elem-def-auto

assume
```

```
    { sublist(cons(t,ts),us) = false,
      elem(t,us) = true }
    elem-sublist
call auto-strategy  elem-sublist
call activate-lemma  elem-sublist

assume
    { subterm_tl(u,ts) = true,
      elem(u,ts) =/= true }
    subterm_tl-elem
call auto-strategy  subterm_tl-elem
call activate-lemma  subterm_tl-elem

assume
    { subterm(u,F(f,ts)) = true,
      elem(u,ts) =/= true }
    subterm-elem
call simplify  subterm-elem
call activate-lemma  subterm-elem

declare operators
    replace1_tl : Termlist Nat Term --> Termlist
    .
assert
  replace1_tl-1 :
    replace1_tl(cons(t,ts),s(0),u) = cons(u,ts)
  replace1_tl-2 :
    replace1_tl(cons(t,ts),s(s(n)),u) = cons(t,replace1_tl(ts,s(n),u))
    .
call analyze-operator  replace1_tl
assume
    { def replace1_tl(ts,n,u),
      n = 0,
      +(1,length(ts)) <= n }
    def-replace1_tl
call auto-strategy  def-replace1_tl
call activate-lemma  def-replace1_tl

assume
    { Well_tl(replace1_tl(ts,n,u)) = Well(u),
      Well_tl(ts) =/= true,
      n = 0,
      +(1,length(ts)) <= n }
    Well_tl-replace1_tl
call auto-strategy  Well_tl-replace1_tl
call activate-lemma  Well_tl-replace1_tl

assume
    { replace1_tl(ts,n,u) =/= replace1_tl(ts,n,v),
      n = 0,
      +(1,length(ts)) <= n,
      u = v }
    replace1_tl-id
call auto-strategy  replace1_tl-id
call activate-lemma  replace1_tl-id
```

```
assume
    { length(replace1_tl(ts,n,u)) = length(ts),
      n = 0,
      +(1,length(ts)) <= n }
    length-replace1_tl
call auto-strategy  length-replace1_tl
call activate-lemma  length-replace1_tl

assume
    { elem(t,replace1_tl(ts,n,u)) = true,
      elem(t,replace1_tl(ts,n,v)) =/= true,
      t = v,
      n = 0,
      +(1,length(ts)) <= n }
    elem-replace1_tl
call auto-strategy  elem-replace1_tl
call activate-lemma  elem-replace1_tl

assume
    { elem(u,replace1_tl(ts,n,u)) = true,
      n = 0,
      +(1,length(ts)) <= n }
    elem-replace1_tl-1
call auto-strategy  elem-replace1_tl-1
call activate-lemma  elem-replace1_tl-1

assume
    { sublist(cons(t,us),replace1_tl(ts,n,v)) =/= true,
      elem(t,replace1_tl(ts,n,u)) = true,
      t = v,
      n = 0,
      +(1,length(ts)) <= n }
    elem-sublist-replace1_tl
call simplify  elem-sublist-replace1_tl
call activate-lemma  elem-sublist-replace1_tl

assume
    { Well(term-arg_tl(ts,n)) = true,
      Well_tl(ts) =/= true,
      n = 0,
      +(1,length(ts)) <= n }
    Well-term-arg_tl
call auto-strategy  Well-term-arg_tl
call activate-lemma  Well-term-arg_tl

declare operators
    replace : Term Position Term --> Term
    .
declare operators
    replace_tl : Termlist Nat Position Term --> Termlist
    .
assert
  replace-1 :
    replace(t,nnil,u) = u
  replace-2 :
    replace(F(f,ts),ncons(n,l),u) = F(f,replace_tl(ts,n,l,u))
```

```
  replace_tl-1 :
    replace_tl(cons(t,ts),s(0),l,u) = cons(replace(t,l,u),ts)
  replace_tl-2 :
    replace_tl(cons(t,ts),s(s(n)),l,u) = cons(t,replace_tl(ts,s(n),l,u))
    .
call analyze-operator  replace
call analyze-operator  replace_tl
assume
    { def replace(t,l,u),
      Well(t) =/= true,
      pos-p(l,t) =/= true }
    def-replace
assume
    { def replace_tl(ts,n,l,u),
      Well_tl(ts) =/= true,
      n = 0,
      +(1,length(ts)) <= n,
      pos-p(l,term-arg_tl(ts,n)) =/= true }
    def-replace_tl
set weight t def-replace
set weight ts def-replace_tl
call auto-strategy  :ind-lemmas { def-replace_tl def-replace } def-replace
call auto-strategy  :ind-lemmas { def-replace def-replace_tl } def-replace_tl
call activate-lemmas  { def-replace def-replace_tl }

assume
    { length(replace_tl(ts,n,l,u)) = length(ts),
      Well_tl(ts) =/= true,
      n = 0,
      +(1,length(ts)) <= n,
      pos-p(l,term-arg_tl(ts,n)) =/= true }
    length-replace_tl
call auto-strategy  length-replace_tl
call activate-lemma  length-replace_tl

assume
    { Well(replace(t,l,u)) = Well(u),
      Well(t) =/= true,
      pos-p(l,t) =/= true }
    Well-replace
assume
    { Well_tl(replace_tl(ts,n,l,u)) = Well(u),
      Well_tl(ts) =/= true,
      n = 0,
      +(1,length(ts)) <= n,
      pos-p(l,term-arg_tl(ts,n)) =/= true }
    Well_tl-replace_tl
set weight t Well-replace
set weight ts Well_tl-replace_tl
call auto-strategy  :ind-lemmas { Well_tl-replace_tl Well-replace } Well-replace
call auto-strategy  :ind-lemmas { Well-replace Well_tl-replace_tl } Well_tl-replace_tl
call activate-lemmas  { Well-replace Well_tl-replace_tl }

assume
    { replace_tl(ts,n,l,u) = replace1_tl(ts,n,replace(term-arg_tl(ts,n),l,u)),
      Well_tl(ts) =/= true,
```

```
      Well(u) =/= true,
      n = 0,
      +(1,length(ts)) <= n,
      pos-p(l,term-arg_tl(ts,n)) =/= true }
    replace_tl-replace1_tl
call auto-strategy  replace_tl-replace1_tl
call activate-lemma  replace_tl-replace1_tl :head-litnbs { 1 } :obl-litnbs-list {  }

assume
    { subterm(ti,F(f,ts)) = true,
      elem(ti,ts) =/= true,
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true }
    arg-subterm
call auto-strategy arg-subterm
call activate-lemmas {arg-subterm}

assume
    { Well(ti) = true,
      Well_tl(ts) =/= true,
      elem(ti,ts) =/= true }
    elem-Well_tl
call auto-strategy elem-Well_tl
call activate-lemmas {elem-Well_tl}

assume
    { Well(ti) = true,
      elem(ti,ts) =/= true,
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true }
    arg-Well
call auto-strategy arg-Well
call activate-lemmas {arg-Well}

assume
    { length(cons(t,ts)) =/= length(nil) }
    length-length
call auto-strategy length-length
call activate-lemma length-length

assume
    { Well_tl(ts1) = true,
      sublist(ts1,ts) =/= true,
      Well_tl(ts) =/= true}
    sublist-Well_tl
call auto-strategy sublist-Well_tl
call activate-lemma sublist-Well_tl :obl-litnbs-list { { 3 } }

declare operators
    term-size : Term --> Nat
    term-size_tl : Termlist --> Nat
    .
assert
  term-size-1 :
    term-size(V(x)) = 1
  term-size-2 :
```

```
    term-size(F(f,ts)) = s(term-size_tl(ts))
  term-size_tl-1 :
    term-size_tl(nil) = 1
  term-size_tl-2 :
    term-size_tl(cons(t,ts)) = +(term-size(t),term-size_tl(ts))
    .
call analyze-operator  term-size
// analyze-operator generates the domain lemma
// { def term-size(t) } named term-size-def-auto
call analyze-operator  term-size_tl
// analyze-operator generates the domain lemma
// { def term-size_tl(ts) } named term-size_tl-def-auto
set weight t term-size-def-auto
set weight ts term-size_tl-def-auto
call auto-strategy  :ind-lemmas { term-size-def-auto term-size_tl-def-auto } ..
                    term-size-def-auto
call auto-strategy  :ind-lemmas { term-size-def-auto term-size_tl-def-auto } ..
                    term-size_tl-def-auto
call activate-lemmas  { term-size-def-auto term-size_tl-def-auto }

assume
    { 1 <= term-size(t) }
    term-size-pos
call auto-strategy  term-size-pos
call activate-linear-lemma  term-size-pos { 1 } { [2] }

assume
    { 1 <= term-size_tl(ts) }
    term-size_tl-pos
call auto-strategy  term-size_tl-pos
call activate-linear-lemma  term-size_tl-pos { 1 } { [2] }

assume
    { term-size_tl(ts) <= term-size_tl(us),
      sublist(ts,us) =/= true }
    term-size_tl-sublist
call auto-strategy  :allow-simplification-before-induction-p FALSE term-size_tl-sublist
call activate-linear-lemma  term-size_tl-sublist { 1 1 } { [1] [2] }

assume
    { ts < us,
      ts = us,
      sublist(ts,us) =/= true }
    sublist-ind
call variables-strategy  { ts us } sublist-ind
call activate-lemma  sublist-ind

assume
    { us < F(f,ts),
      sublist(us,ts) =/= true }
    F-sublist-ind
apply lit-add
    us = ts
    . ..
    F-sublist-ind
call simplify  F-sublist-ind
```

```
apply <-trans 2 ts F-sublist-ind
call simplify  F-sublist-ind
call simplify  F-sublist-ind
call deactivate-lemmas { sublist-ind }

assume
    { t < u,
      subterm(t,u) =/= true }
    subterm-ind
assume
    { t < F(g,us),
      subterm_tl(t,us) =/= true }
    subterm_tl-ind
set weight u subterm-ind
set weight us subterm_tl-ind
call auto-strategy  :ind-lemmas { subterm-ind subterm_tl-ind } subterm-ind
call auto-strategy  :ind-lemmas { subterm-ind subterm_tl-ind } subterm_tl-ind
apply <-trans 4 F(g,us) subterm_tl-ind
call simplify  :ind-lemmas { subterm-ind subterm_tl-ind } subterm_tl-ind
call simplify  subterm_tl-ind
apply <-trans 3 u subterm_tl-ind
call simplify  :ind-lemmas { subterm-ind subterm_tl-ind } subterm_tl-ind
call simplify  subterm_tl-ind
```

A.7 Lpo

This file contains the internal representation Lpo of the lexicographic path order. First, we prove general auxiliary lemmas such as the transitivity, irreflexivity and containedness of the subterm relation. Then, we consider specialized lemmas required for the equivalence proofs between the different variants.

```
declare operators
    Lpo : Term Term --> Bool
    Alpha : Termlist Term --> Bool
    Beta : Term Term --> Bool
    Gamma : Term Term --> Bool
    Delta : Term Term --> Bool
    Majo : Term Termlist --> Bool
    Lex : Termlist Termlist --> Bool
    .
assert
  Lpo-1 :
    Lpo(F(f,ts),F(g,us)) = true
      if Alpha(ts,F(g,us)) = true
  Lpo-2 :
    Lpo(F(f,ts),F(g,us)) = true if Beta(F(f,ts),F(g,us)) = true
  Lpo-3 :
    Lpo(F(f,ts),F(g,us)) = Gamma(F(f,ts),F(g,us))
      if Alpha(ts,F(g,us)) =/= true,
         def Alpha(ts,F(g,us)),
         Beta(F(f,ts),F(g,us)) =/= true,
         def Beta(F(f,ts),F(g,us))
  Lpo-4 :
```

```
  Lpo(F(f,ts),V(y)) = Delta(F(f,ts),V(y))
Lpo-5 :
  Lpo(V(x),u) = false
Alpha-1 :
  Alpha(nil,u) = false
Alpha-2 :
  Alpha(cons(t,ts),u) = true
    if t = u
Alpha-3 :
  Alpha(cons(t,ts),u) = true
    if t =/= u,
       Lpo(t,u) = true
Alpha-4 :
  Alpha(cons(t,ts),u) = Alpha(ts,u)
    if t =/= u,
       Lpo(t,u) =/= true,
       def Lpo(t,u)
Beta-1 :
  Beta(F(f,ts),F(g,us)) = Majo(F(f,ts),us)
    if prec(f,g) = true
Beta-2 :
  Beta(F(f,ts),F(g,us)) = false
    if prec(f,g) =/= true,
       def prec(f,g)
Gamma-1 :
  Gamma(F(f,ts),F(g,us)) = Majo(F(f,ts),us)
    if f = g,
       Lex(ts,us) = true
Gamma-2 :
  Gamma(F(f,ts),F(g,us)) = false
    if f =/= g
Gamma-3 :
  Gamma(F(f,ts),F(g,us)) = false
    if f = g,
       Lex(ts,us) =/= true,
       def Lex(ts,us)
Delta-1 :
  Delta(F(f,ts),V(y)) = contains_tl(ts,y)
Majo-1 :
  Majo(t,nil) = true
Majo-2 :
  Majo(t,cons(u,us)) = Majo(t,us)
    if Lpo(t,u) = true
Majo-3 :
  Majo(t,cons(u,us)) = false
    if Lpo(t,u) =/= true,
       def Lpo(t,u)
Lex-1 :
  Lex(nil,nil) = false
Lex-2 :
  Lex(cons(t,ts),cons(u,us)) = Lex(ts,us)
    if t = u
Lex-3 :
  Lex(cons(t,ts),cons(u,us)) = Lpo(t,u)
    if t =/= u
  .
```

```
call analyze-operator  Lpo :speculate-domain-lemma-p FALSE
call analyze-operator  Alpha :speculate-domain-lemma-p FALSE
call analyze-operator  Beta :speculate-domain-lemma-p FALSE
call analyze-operator  Gamma :speculate-domain-lemma-p FALSE
call analyze-operator  Delta :speculate-domain-lemma-p FALSE
call analyze-operator  Majo :speculate-domain-lemma-p FALSE
call analyze-operator  Lex :speculate-domain-lemma-p FALSE
assume
    { def Lpo(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    Lpo-def-manuell
assume
    { def Alpha(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    Alpha-def-manuell
assume
    { def Beta(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    Beta-def-manuell
assume
    { def Gamma(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    Gamma-def-manuell
assume
    { def Delta(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    Delta-def-manuell
assume
    { def Majo(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    Majo-def-manuell
assume
    { def Lex(ts,us),
      length(ts) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true  }
    Lex-def-manuell
set weight (t,u,0) Lpo-def-manuell
set weight (ts,t) Alpha-def-manuell
set weight (t,u) Beta-def-manuell
set weight (t,u) Gamma-def-manuell
set weight (ts,us) Lex-def-manuell
set weight (t,us) Majo-def-manuell
call operators-strategy { 1 } { [1] } Delta-def-manuell
```

```
call activate-lemma  Delta-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { Alpha-def-manuell
         Beta-def-manuell Gamma-def-manuell } Lpo-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { Lpo-def-manuell
         Alpha-def-manuell } Alpha-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { Majo-def-manuell
         Beta-def-manuell } Beta-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { Majo-def-manuell
         Lex-def-manuell Gamma-def-manuell } Gamma-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { Lpo-def-manuell
         Majo-def-manuell } Majo-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { Lpo-def-manuell
         Lex-def-manuell } Lex-def-manuell
call activate-lemmas { Lpo-def-manuell
                       Alpha-def-manuell
                       Beta-def-manuell
                       Gamma-def-manuell
                       Majo-def-manuell
                       Lex-def-manuell }

assume
    { Lpo(t,u) = false,
      contains(t,y) = true,
      contains(u,y) =/= true,
      Well(t) =/= true,
      Well(u) =/= true  }
    contains-Lpo
assume
    { Alpha(ts,t) = false,
      contains_tl(ts,y) = true,
      contains(t,y) =/= true,
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    contains-Alpha
assume
    { Beta(t,u) = false,
      contains(t,y) = true,
      contains(u,y) =/= true,
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    contains-Beta
assume
    { Gamma(t,u) = false,
      contains(t,y) = true,
      contains(u,y) =/= true,
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    contains-Gamma
assume
    { Delta(t,u) = false,
      contains(t,y) = true,
      contains(u,y) =/= true,
```

```
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    contains-Delta
assume
    { Majo(t,us) = false,
      contains(t,y) = true,
      contains_tl(us,y) =/= true,
      Well(t) =/= true,
      Well_tl(us) =/= true }
    contains-Majo
set weight (t,u,0) contains-Lpo
set weight (ts,t) contains-Alpha
set weight (t,u) contains-Beta
set weight (t,u) contains-Gamma
set weight (t,us) contains-Majo
call operators-strategy { 1 } { [1] } contains-Delta
call activate-lemma  contains-Delta
call operators-strategy { 1 } { [1] } :ind-lemmas { contains-Alpha
         contains-Beta contains-Gamma contains-Lpo } contains-Lpo
call operators-strategy { 1 } { [1] } :ind-lemmas { contains-Lpo
         contains-Alpha } contains-Alpha
call operators-strategy { 1 } { [1] } :ind-lemmas { contains-Lpo
         contains-Majo contains-Beta } contains-Beta
call operators-strategy { 1 } { [1] } :ind-lemmas { contains-Lpo
         contains-Majo contains-Gamma } contains-Gamma
call operators-strategy { 1 } { [1] } :ind-lemmas { contains-Lpo
         contains-Majo } contains-Majo
call activate-lemmas { contains-Lpo
                       contains-Alpha
                       contains-Beta
                       contains-Gamma
                       contains-Majo }

assume
    { Lpo(F(f,ts),u) = true,
      Alpha(ts,u) =/= true,
      Well_tl(ts) =/= true,
      arity(f) =/= length(ts),
      Well(u) =/= true,
      Fun(u) =/= true  }
    Lpo-Alpha
call variables-strategy { u } Lpo-Alpha

assume
    { Lpo(t,u) = true,
      Beta(t,u) =/= true,
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    Lpo-Beta
call auto-strategy  Lpo-Beta

assume
```

```
    { Lpo(t,u) = true,
      Gamma(t,u) =/= true,
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    Lpo-Gamma
call auto-strategy  Lpo-Gamma

assume
    { Lpo(u,t) = true,
      subterm(t,u) =/= true,
      Well(t) =/= true,
      Well(u) =/= true }
    Lpo-subterm
assume
    { Alpha(us,t) = true,
      subterm_tl(t,us) =/= true,
      Well(t) =/= true,
      Well_tl(us) =/= true }
    Alpha-subterm_tl
set weight u Lpo-subterm
set weight us Alpha-subterm_tl
call variables-strategy  { u t } :ind-lemmas { Alpha-subterm_tl } Lpo-subterm
call auto-strategy :ind-lemmas { Lpo-subterm Alpha-subterm_tl } Alpha-subterm_tl
call activate-lemma  Alpha-subterm_tl

assume
    { Lpo(F(g,us),t) = true,
      subterm_tl(t,us) =/= true,
      Well(t) =/= true,
      Well_tl(us) =/= true,
      arity(g) =/= length(us) }
    Lpo-subterm_tl
call variables-strategy  { t } Lpo-subterm_tl
call activate-lemma  Lpo-subterm_tl

assume
    { Lpo(t,v) =/= true,
      Lpo(v,u) =/= true,
      Lpo(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(v) =/= true }
    Lpo-trans
assume
    { Alpha(ts,F(h,vs)) =/= true,
      Alpha(vs,u) =/= true,
      Alpha(ts,u) = true,
      Well_tl(ts) =/= true,
      Well(u) =/= true,
      Well_tl(vs) =/= true,
      arity(h) =/= length(vs),
      Fun(u) =/= true }
    Alpha-Alpha-trans
assume
```

```
    { Alpha(ts,v) =/= true,
      Beta(v,u) =/= true,
      Alpha(ts,u) = true,
      Well_tl(ts) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      Fun(u) =/= true,
      Fun(v) =/= true }
    Alpha-Beta-trans
assume
    { Alpha(ts,v) =/= true,
      Gamma(v,u) =/= true,
      Alpha(ts,u) = true,
      Well_tl(ts) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      Fun(u) =/= true,
      Fun(v) =/= true }
    Alpha-Gamma-trans
assume
    { Beta(t,F(h,vs)) =/= true,
      Alpha(vs,u) =/= true,
      Lpo(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(vs) =/= true,
      arity(h) =/= length(vs),
      Fun(t) =/= true,
      Fun(u) =/= true  }
    Beta-Alpha-trans
assume
    { Beta(t,v) =/= true,
      Beta(v,u) =/= true,
      Beta(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true,
      Fun(v) =/= true  }
    Beta-Beta-trans
assume
    { Beta(t,v) =/= true,
      Gamma(v,u) =/= true,
      Beta(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true,
      Fun(v) =/= true  }
    Beta-Gamma-trans
assume
    { Gamma(t,F(h,vs)) =/= true,
      Alpha(vs,u) =/= true,
      Lpo(t,u) = true,
```

```
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(vs) =/= true,
      arity(h) =/= length(vs),
      Fun(t) =/= true,
      Fun(u) =/= true  }
    Gamma-Alpha-trans
assume
    { Gamma(t,v) =/= true,
      Beta(v,u) =/= true,
      Beta(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true,
      Fun(v) =/= true  }
    Gamma-Beta-trans
assume
    { Gamma(t,v) =/= true,
      Gamma(v,u) =/= true,
      Gamma(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true,
      Fun(v) =/= true  }
    Gamma-Gamma-trans
assume
    { Majo(t,vs) =/= true,
      Alpha(vs,u) =/= true,
      Lpo(t,u) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(vs) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    Majo-Alpha-trans
assume
    { Majo(F(f,ts),us) =/= true,
      Majo(F(g,us),vs) =/= true,
      Majo(F(f,ts),vs) = true,
      prec(f,g) =/= true,
      Well_tl(ts) =/= true,
      arity(f) =/= length(ts),
      Well_tl(us) =/= true,
      arity(g) =/= length(us),
      Well_tl(vs) =/= true }
    Majo-Majo-trans
assume
    { Majo(F(g,ts),us) =/= true,
      Majo(F(g,us),vs) =/= true,
      Majo(F(g,ts),vs) = true,
      Lex(ts,us) =/= true,
      Well_tl(ts) =/= true,
```

```
      arity(g) =/= length(ts),
      Well_tl(us) =/= true,
      arity(g) =/= length(us),
      Well_tl(vs) =/= true }
    Majo-Majo-Lex-trans
assume
    { Lex(ts,us) =/= true,
      Lex(us,vs) =/= true,
      Lex(ts,vs) = true,
      length(ts) =/= length(us),
      length(ts) =/= length(vs),
      Well_tl(ts) =/= true,
      Well_tl(vs) =/= true,
      Well_tl(us) =/= true }
    Lex-Lex-trans
assume
    { Alpha(ts,u) =/= true,
      Delta(u,V(y)) =/= true,
      Delta(F(f,ts),V(y)) = true,
      Well_tl(ts) =/= true,
      arity(f) =/= length(ts),
      Well(u) =/= true,
      Well(V(y)) =/= true,
      Fun(u) =/= true }
    Alpha-Delta-trans
assume
    { Beta(t,u) =/= true,
      Delta(u,V(y)) =/= true,
      Delta(t,V(y)) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(V(y)) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    Beta-Delta-trans
assume
    { Gamma(t,u) =/= true,
      Delta(u,V(y)) =/= true,
      Delta(t,V(y)) = true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(V(y)) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    Gamma-Delta-trans
assume
    { Lpo(t,t) = false,
      Well(t) =/= true }
    Lpo-irrefl
assume
    { Alpha(ts,F(g,us)) = false,
      sublist(ts,us) =/= true,
      Well_tl(us) =/= true,
      arity(g) =/= length(us),
      Well_tl(ts) =/= true }
    Alpha-irrefl
```

```
assume
    { Beta(t,t) = false,
      Well(t) =/= true,
      Fun(t) =/= true }
    Beta-irrefl
assume
    { Gamma(t,t) = false,
      Well(t) =/= true,
      Fun(t) =/= true }
    Gamma-irrefl
assume
    { Lex(ts,ts) = false,
      Well_tl(ts) =/= true }
    Lex-irrefl
call set-default-settings :activate-first-lit-p TRUE
call auto-strategy  Beta-irrefl
call activate-lemma  Beta-irrefl
call auto-strategy  Lex-irrefl
call activate-lemma  Lex-irrefl
call auto-strategy  Gamma-irrefl
call activate-lemma  Gamma-irrefl
call simplify  Alpha-Delta-trans
call activate-lemma  Alpha-Delta-trans
call operators-strategy { 1 } { [1] } Beta-Delta-trans
call activate-lemma  Beta-Delta-trans
call operators-strategy { 1 } { [1] } Gamma-Delta-trans
call activate-lemma  Gamma-Delta-trans
set weight (t,v,u,0) Lpo-trans
set weight (ts) Alpha-Alpha-trans
set weight (ts) Alpha-Beta-trans
set weight (ts) Alpha-Gamma-trans
set weight (t,F(h,vs)) Beta-Alpha-trans
set weight (t,v,u) Beta-Beta-trans
set weight (t,v,u) Beta-Gamma-trans
set weight (t,F(h,vs)) Gamma-Alpha-trans
set weight (t,v,u) Gamma-Beta-trans
set weight (t,v,u) Gamma-Gamma-trans
set weight (t,vs) Majo-Alpha-trans
set weight (F(f,ts),F(g,us),vs) Majo-Majo-trans
set weight (F(g,ts),F(g,us),vs) Majo-Majo-Lex-trans
set weight (ts) Lex-Lex-trans
set weight (t) Lpo-irrefl
set weight (ts) Alpha-irrefl
call deactivate-axioms  { Alpha-1 Alpha-2 Alpha-3 Alpha-4 Beta-1 Beta-2
                          Gamma-1 Gamma-2 Gamma-3 Delta-1 }
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Alpha-Alpha-trans
         Alpha-Beta-trans Alpha-Gamma-trans Beta-Alpha-trans Beta-Beta-trans
         Beta-Gamma-trans Gamma-Alpha-trans Gamma-Beta-trans Gamma-Gamma-trans } ..
     Lpo-trans
call activate-axioms  { Delta-1 Gamma-3 Gamma-2 Gamma-1 Beta-2 Beta-1
                        Alpha-4 Alpha-3 Alpha-2 Alpha-1 }
call activate-lemma  Lpo-Alpha
call operators-strategy  { 1 } { [1] } :ind-lemmas { Lpo-trans
         Alpha-Alpha-trans } Alpha-Alpha-trans
call deactivate-lemmas { Lpo-Alpha }
call activate-lemma  Lpo-Beta
```

```
call operators-strategy  { 1 } { [1] } :ind-lemmas { Lpo-trans
         Alpha-Beta-trans } Alpha-Beta-trans
call deactivate-lemmas { Lpo-Beta }
call activate-lemma  Lpo-Gamma
call operators-strategy  { 1 } { [1] } :ind-lemmas { Lpo-trans
         Alpha-Gamma-trans } Alpha-Gamma-trans
call deactivate-lemmas { Lpo-Gamma }
call variables-strategy { t } :ind-lemmas { Majo-Alpha-trans } Beta-Alpha-trans
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Majo-Majo-trans
         Beta-Beta-trans } Beta-Beta-trans
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Majo-Majo-trans
         Beta-Gamma-trans } Beta-Gamma-trans
call variables-strategy { t } :ind-lemmas { Majo-Alpha-trans } Gamma-Alpha-trans
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Majo-Majo-Lex-trans
         Gamma-Beta-trans } Gamma-Beta-trans
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Majo-Majo-Lex-trans
         Lex-Lex-trans Gamma-Gamma-trans } Gamma-Gamma-trans
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Lpo-trans
         Majo-Alpha-trans } Majo-Alpha-trans
call operators-strategy  { 2 } { [1] } :ind-lemmas { Lpo-trans
         Majo-Majo-trans } Majo-Majo-trans
apply ind-subs
    Lpo-trans
    [t<--F(f,ts),v <-- F(g,us) , u <-- u] ..
    Majo-Majo-trans
call cont-proof-attempt :ind-lemmas { Majo-Majo-trans } Majo-Majo-trans
call operators-strategy  { 2 } { [1] } :ind-lemmas { Lpo-trans
         Majo-Majo-Lex-trans } Majo-Majo-Lex-trans
apply ind-subs
    Lpo-trans
    [t <-- F(g,ts) , u <-- u, v<--F(g,us)] ..
    Majo-Majo-Lex-trans
call cont-proof-attempt :ind-lemmas { Majo-Majo-Lex-trans } Majo-Majo-Lex-trans
call activate-lemma  Lpo-subterm
call operators-strategy  { 1 2 } { [1] [1] } :ind-lemmas { Lpo-trans
         Lpo-irrefl Lex-Lex-trans } Lex-Lex-trans
call deactivate-lemmas { Lpo-subterm }
call auto-strategy :ind-lemmas { Alpha-irrefl Lpo-irrefl } Lpo-irrefl
call auto-strategy  Alpha-irrefl
apply lemma-rewrite
    4
    [1]
    subterm_tl-sublist
    1
    [u <-- u , ts <-- ts , us <-- us] ..
    Alpha-irrefl
apply ind-subs
    Lpo-trans
    [t <-- u , v <-- F(g,us)] ..
    Alpha-irrefl
call activate-lemma  Lpo-subterm
call cont-proof-attempt :ind-lemmas { Lpo-irrefl } Alpha-irrefl
call deactivate-lemmas { Lpo-subterm }
call deactivate-lemmas  { Alpha-Delta-trans Beta-Delta-trans Gamma-Delta-trans }
call activate-lemma Lpo-trans :head-litnbs { 3 } :obl-litnbs-list { { 1 } { 2 } }
call activate-lemma Alpha-Alpha-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
```

```
call activate-lemma Alpha-Beta-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Alpha-Gamma-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Beta-Alpha-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Beta-Beta-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Beta-Gamma-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Gamma-Alpha-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Gamma-Beta-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Gamma-Gamma-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Majo-Alpha-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Majo-Majo-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 4 } }
call activate-lemma Majo-Majo-Lex-trans :head-litnbs { 1 2 } ..
                    :obl-litnbs-list { { 1 2 4 } }
call activate-lemma Lex-Lex-trans :head-litnbs { 1 2 } :obl-litnbs-list { { 1 2 } }
call activate-lemma Lpo-irrefl
call activate-lemma Alpha-irrefl :obl-litnbs-list { { 2 } }
call set-default-settings :activate-first-lit-p FALSE

assume
    { Lpo(t,u) =/= true,
      Lpo(u,t) =/= true,
      Well(t) =/= true,
      Well(u) =/= true }
    Lpo-irrefl-trans
apply lemma-subs
    Lpo-trans
    [t <-- t , v <-- u, u <-- t] ..
    Lpo-irrefl-trans
call simplify  Lpo-irrefl-trans
call activate-lemma  Lpo-irrefl-trans :head-litnbs { 1 } :obl-litnbs-list { { 2 } }

assume
    { Lex(replace1_tl(ts,n,u),replace1_tl(ts,n,v)) = true,
      Lpo(u,v) =/= true,
      Well_tl(ts) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      n = 0,
      +(1,length(ts)) <= n }
    Lex-replace1_tl
call operators-strategy  { 1 1 } { [1:1] [1:2] }  Lex-replace1_tl
call activate-lemma  Lex-replace1_tl

assume
    { Majo(F(f,replace1_tl(ts,n,u)),us) = true,
      sublist(us,replace1_tl(ts,n,v)) =/= true,
      Lpo(u,v) =/= true,
      Well_tl(ts) =/= true,
      arity(f) =/= length(ts),
      Well_tl(us) =/= true,
      Well(u) =/= true,
      n = 0,
      +(1,arity(f)) <= n }
    Majo-replace1_tl
call activate-lemma  Lpo-subterm :head-litnbs { 1 } :obl-litnbs-list {  }
call activate-lemma  subterm_tl-elem :head-litnbs { 1 } :obl-litnbs-list {  }
call auto-strategy  Majo-replace1_tl
```

```
call activate-lemma  Majo-replace1_tl
call activate-lemma  Lpo-subterm
call activate-lemma  subterm_tl-elem

assume
    { Gamma(F(f,replace1_tl(ts,n,u)),F(f,replace1_tl(ts,n,v))) = true,
      Lpo(u,v) =/= true,
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      n = 0,
      +(1,length(ts)) <= n }
    Gamma-replace1_tl
call simplify  Gamma-replace1_tl
call activate-lemma  Gamma-replace1_tl

assume
    { Lpo(F(f,replace1_tl(ts,n,u)),F(f,replace1_tl(ts,n,v))) = true,
      Lpo(u,v) =/= true,
      Well_tl(ts) =/= true,
      arity(f) =/= length(ts),
      Well(u) =/= true,
      Well(v) =/= true,
      n = 0,
      +(1,length(ts)) <= n }
    Lpo-replace1_tl
call simplify  Lpo-replace1_tl
call activate-lemma  Lpo-replace1_tl

assume
    { Lpo(replace(t,l,u),replace(t,l,v)) =  true,
      Lpo(u,v) =/= true,
      Well(t) =/= true,
      Well(u) =/= true,
      Well(v) =/= true,
      pos-p(l,t) =/= true }
    Lpo-replace
call deactivate-axioms  { Lpo-1 Lpo-2 Lpo-3 Lpo-4 Lpo-5 }
call operators-strategy  { 1 1 } { [1:1] [1:2] }  Lpo-replace
call activate-axioms  { Lpo-5 Lpo-4 Lpo-3 Lpo-2 Lpo-1 }
call activate-lemma  Lpo-replace

declare operators
    Lpoeq : Term Term --> Bool
    .
assert
  Lpoeq-1 :
    Lpoeq(t,u) = true
      if t = u
  Lpoeq-2 :
    Lpoeq(t,u) = Lpo(t,u)
      if t =/= u
    .
call analyze-operator  Lpoeq :speculate-domain-lemma-p FALSE
call activate-axiom  Lpoeq-2 :obl-litnbs-list {  }
```

```
call activate-axiom  Lpoeq-1 :obl-litnbs-list {  }

assume
    { def Lpoeq(t,u),
      Well(t) =/= true,
      Well(u) =/= true }
    Lpoeq-def-manuell
call operators-strategy  { 1 } { [1] } Lpoeq-def-manuell
call activate-lemma  Lpoeq-def-manuell

assume
    { Lpo(t1,t3) = true,
      subterm(t2,t1) =/= true,
      Lpoeq(t2,t3) =/= true,
      Well(t1) =/= true,
      Well(t2) =/= true,
      Well(t3) =/= true }
    Lemma2
call operators-strategy  { 2 3 } { [1] [1] } Lemma2
call activate-lemma  Lemma2 :head-litnbs { 1 } :obl-litnbs-list { { 2 } { 3 } }

assume
    { Lpo(F(f,ts1),u) = true,
      Lpo(t,u) =/= true,
      sublist(cons(t,ts),ts1) =/= true,
      arity(f) =/= length(ts1),
      Well_tl(ts1) =/= true,
      Well(u) =/= true,
      Well(t) =/= true,
      Well_tl(ts) =/= true }
    Lpo-sublist
call simplify  Lpo-sublist

assume
    { Lpo(t,v) = true,
      subterm(u,t) =/= true,
      Lpo(u,v) =/= true,
      Well(t) =/= true,
      Well(v) =/= true }
     subterm-Lpo-trans
call auto-strategy subterm-Lpo-trans
call activate-lemma subterm-Lpo-trans :obl-litnbs-list { { 2 } { 3 } }

assume
    { Lpo(t,v) = true,
      subterm(u,t) =/= true,
      Lpoeq(u,v) =/= true,
      Well(t) =/= true,
      Well(v) =/= true }
     subterm-Lpoeq-trans
call auto-strategy subterm-Lpoeq-trans
call activate-lemma subterm-Lpoeq-trans

assume
    { Lpo(t,v) = true,
      subterm(v,u) =/= true,
```

```
      Lpo(t,u) =/= true,
      Well(t) =/= true,
      Well(u) =/= true }
     Lpo-subterm-trans
call auto-strategy Lpo-subterm-trans

assume
    { Lpo(F(f,ts),ti) = true,
      elem(ti,ts) =/= true,
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true }
    arg-Lpo
call auto-strategy arg-Lpo
call activate-lemma arg-Lpo


assume
    { Lpo(t,w) = true,
      subterm(u,t) =/= true,
      Lpoeq(u,v) =/= true,
      subterm(w,v) =/= true,
      Well(t) =/= true,
      Well(v) =/= true }
    subterm-Lpo-subterm-trans
call simplify  :allow-alternative-free-var-bindings-p TRUE subterm-Lpo-subterm-trans
call activate-lemma subterm-Lpo-subterm-trans

assume
    { Lpo(F(f,ts),ui) = true,
      elem(ti,ts) =/= true,
      elem(ui,us) =/= true,
      Lpoeq(ti, F(g,us)) =/= true,
      arity(f) =/= length(ts),
      arity(g) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    Lemma3
apply lemma-rewrite
    1
    [1]
    subterm-Lpo-subterm-trans
    1
    [t <-- F(f,ts) , w <-- ui, u <-- ti, v <-- F(g,us)] ..
    Lemma3
call cont-proof-attempt  Lemma3

assume
    { Lpoeq(u,t) =/= true,
      Well(t) =/= true,
      Well(u) =/= true,
      Lpo(t,u) =/= true}
    Lpo-implies-notLpoeq-converse
call simplify  Lpo-implies-notLpoeq-converse
call activate-lemma Lpo-implies-notLpoeq-converse

assume
```

```
    { Alpha(cons(t,ts),w) = Alpha(ts,w),
      Lpo(w,t) =/= true,
      Well(t) =/= true,
      Well_tl(ts) =/= true,
      Well(w) =/= true }
    Alpha-chop
call simplify  Alpha-chop
call activate-lemma Alpha-chop

assume
    { Alpha(cons(t,ts),F(g,us)) = Alpha(ts,F(g,us)),
      sublist(cons(u,us1),us) =/= true,
      Lpo(t,u) = true,
      t = u,
      arity(g) =/= length(us),
      Well(u) =/= true,
      Well(t) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true,
      Well_tl(us1) =/= true }
    Alpha-chop1
call simplify  Alpha-chop1
call activate-lemma Alpha-chop1

assume
    { Alpha(ts,u) = true,
      sublist(us,ts) =/= true,
      Alpha(us,u) =/= true,
      Well(u) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    Alpha-sublist
call auto-strategy  Alpha-sublist
call activate-lemma  Alpha-sublist :obl-litnbs-list { { 2 } }

assume
    { Lpo(F(f,ts),ui) = true,
      sublist(ts1,ts) =/= true,
      elem(ui,us) =/= true,
      Alpha(ts1,F(g,us)) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true,
      arity(f) =/= length(ts),
      arity(g) =/= length(us) }
    Lemma3-consequence-2
apply lemma-subs
    Lpo-trans
    [t <-- F(f,ts), v <-- F(g,us) , u <-- ui] ..
    Lemma3-consequence-2
call cont-proof-attempt  :ind-lemmas { } Lemma3-consequence-2
call activate-lemma Lemma3-consequence-2 :obl-litnbs-list { { 4 } }
call deactivate-lemmas { Alpha-sublist }

assume
    { Majo(F(f,ts),us1) = true,
      Alpha(ts1,F(g,us)) =/= true,
```

```
      sublist(ts1,ts) =/= true,
      sublist(us1,us) =/= true,
      arity(g) =/= length(us),
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true,
      Well_tl(ts1) =/= true,
      Well_tl(us1) =/= true }
    Lemma3-consequence-3
call operators-strategy  { 1 } { [1] } Lemma3-consequence-3
call activate-lemma Lemma3-consequence-3
call deactivate-lemmas { Lemma3-consequence-2 }

assume
    { Majo(F(f,ts),us1) = true,
      Alpha(ts1,F(g,us)) =/= true,
      sublist(ts1,ts) =/= true,
      sublist(us1,us) =/= true,
      arity(g) =/= length(us),
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    Lemma3-consequence-4
call simplify  Lemma3-consequence-4
call activate-lemma Lemma3-consequence-4
call deactivate-lemmas { Lemma3-consequence-3 }

assume
    { or(Alpha(ts1,F(g,us)), Majo(F(f,ts),us1)) = Majo(F(f,ts),us1),
      sublist(ts1,ts) =/= true,
      sublist(us1,us) =/= true,
      arity(f) =/= length(ts),
      arity(g) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    Lemma3-consequence-5
call simplify  :allow-alternative-free-var-bindings-p TRUE Lemma3-consequence-5
call activate-lemma Lemma3-consequence-5
call deactivate-lemmas { Lemma3-consequence-4 }

assume
    { or(Majo(F(f,ts),us1), Alpha(ts1,F(g,us))) = Majo(F(f,ts),us1),
      sublist(ts1,ts) =/= true,
      sublist(us1,us) =/= true,
      arity(f) =/= length(ts),
      arity(g) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    Lemma3-consequence-5a
call simplify  Lemma3-consequence-5a
call activate-lemma Lemma3-consequence-5a

assume
    { or(Alpha(ts,F(g,us)), Majo(F(f,ts),us)) = Majo(F(f,ts),us),
      arity(f) =/= length(ts),
      arity(g) =/= length(us),
```

```
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    Lemma3-consequence-6
call simplify  Lemma3-consequence-6
call activate-lemma Lemma3-consequence-6

call deactivate-lemmas  { Lpo-trans sublist-subterms Lpo-subterm_tl }
```

A.8 lpo1

This file (as well as each of the following files) contains exactly one variant of the LPO and an equivalence proof w.r.t. a previous variant.

```
declare operators
    lpo1 : Term Term --> Bool
    alpha1 : Termlist Term --> Bool
    beta1 : Term Term --> Bool
    gamma1 : Term Term --> Bool
    delta1 : Term Term --> Bool
    majo1 : Term Termlist --> Bool
    lex1 : Termlist Termlist --> Bool
    .
assert
  lex1-1 :
    lex1(nil,nil) = false
  lex1-2 :
    lex1(cons(t,ts),cons(u,us)) = lex1(ts,us)
      if t = u
  lex1-3 :
    lex1(cons(t,ts),cons(u,us)) = lpo1(t,u)
      if t =/= u
  majo1-1 :
    majo1(t,nil) = true
  majo1-2 :
    majo1(t,cons(u,us)) = and(lpo1(t,u),majo1(t,us))
  delta1-1 :
    delta1(F(f,ts),V(y)) = contains_tl(ts,y)
  gamma1-1 :
    gamma1(F(f,ts),F(g,us)) = and(lex1(ts,us),majo1(F(f,ts),us))
      if f = g
  gamma1-2 :
    gamma1(F(f,ts),F(g,us)) = false
    if
      f =/= g
  beta1-1 :
    beta1(F(f,ts),F(g,us)) = and(prec(f,g),majo1(F(f,ts),us))
  alpha1-1 :
    alpha1(nil,u) = false
  alpha1-2 :
    alpha1(cons(t,ts),u) = true
      if t = u
  alpha1-3 :
    alpha1(cons(t,ts),u) = or(lpo1(t,u),alpha1(ts,u))
      if t =/= u
```

```
  lpo1-1 :
    lpo1(V(x),u) = false
  lpo1-2 :
    lpo1(F(f,ts),V(y)) = delta1(F(f,ts),V(y))
  lpo1-3 :
    lpo1(F(f,ts),F(g,us)) = or(alpha1(ts,F(g,us)),or(beta1(F(f,ts),F(g,us)),
                                                  gamma1(F(f,ts),F(g,us))))
    .
call analyze-operator lpo1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator alpha1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator beta1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator gamma1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator delta1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator majo1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator lex1 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lpo1(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo1-def-manuell
assume
    { def alpha1(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha1-def-manuell
assume
    { def beta1(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    beta1-def-manuell
assume
    { def gamma1(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    gamma1-def-manuell
assume
    { def delta1(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta1-def-manuell
assume
    { def majo1(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo1-def-manuell
assume
    { def lex1(ts,us),
      length(ts) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true  }
```

```
    lex1-def-manuell
set weight (t,u,0) lpo1-def-manuell
set weight (ts,t) alpha1-def-manuell
set weight (t,u) beta1-def-manuell
set weight (t,u) gamma1-def-manuell
set weight (ts,us) lex1-def-manuell
set weight (t,us) majo1-def-manuell
call operators-strategy { 1 } { [1] } delta1-def-manuell
call activate-lemma  delta1-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { alpha1-def-manuell
         beta1-def-manuell gamma1-def-manuell } lpo1-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo1-def-manuell
         alpha1-def-manuell } alpha1-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { majo1-def-manuell
         beta1-def-manuell } beta1-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { majo1-def-manuell
         lex1-def-manuell gamma1-def-manuell } gamma1-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo1-def-manuell
         majo1-def-manuell } majo1-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo1-def-manuell
         lex1-def-manuell } lex1-def-manuell
call activate-lemmas { lpo1-def-manuell
                       alpha1-def-manuell
                       beta1-def-manuell
                       gamma1-def-manuell
                       majo1-def-manuell
                       lex1-def-manuell }

assume
    { lpo1(t,u) = Lpo(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo1-Lpo
assume
    { alpha1(ts,t) = Alpha(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha1-Alpha
assume
    { beta1(t,u) = Beta(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    beta1-Beta
assume
    { gamma1(t,u) = Gamma(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    gamma1-Gamma
assume
    { delta1(t,u) = Delta(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
```

```
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta1-Delta
assume
    { majo1(t,us) = Majo(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo1-Majo
assume
    { lex1(ts,us) = Lex(ts,us),
      length(ts) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lex1-Lex
set weight (t,u,0) lpo1-Lpo
set weight (ts,t) alpha1-Alpha
set weight (t,u) beta1-Beta
set weight (t,u) gamma1-Gamma
set weight (ts,us) lex1-Lex
set weight (t,us) majo1-Majo
call operators-strategy { 1 1 } { [1] [2] } delta1-Delta
call activate-lemma  delta1-Delta :obl-litnbs-list { }
call operators-strategy  { 1 1 } { [1] [2] } :ind-lemmas { alpha1-Alpha
         beta1-Beta gamma1-Gamma } lpo1-Lpo
call operators-strategy  { 1 1 } { [1] [2] } :ind-lemmas { alpha1-Alpha
         lpo1-Lpo } alpha1-Alpha
call operators-strategy  { 1 1 } { [1] [2] } :ind-lemmas { beta1-Beta
         majo1-Majo } beta1-Beta
call operators-strategy  { 1 1 } { [1] [2] } :ind-lemmas { majo1-Majo
         lex1-Lex gamma1-Gamma } gamma1-Gamma
call operators-strategy  { 1 1 } { [1] [2] } :ind-lemmas { majo1-Majo
         lpo1-Lpo } majo1-Majo
call operators-strategy  { 1 1 } { [1] [2] } :ind-lemmas { lpo1-Lpo
         lex1-Lex } lex1-Lex
call activate-lemma  lpo1-Lpo :obl-litnbs-list { }
call activate-lemma  alpha1-Alpha :obl-litnbs-list { }
call activate-lemma  beta1-Beta :obl-litnbs-list { }
call activate-lemma  gamma1-Gamma :obl-litnbs-list { }
call activate-lemma  majo1-Majo :obl-litnbs-list { }
call activate-lemma  lex1-Lex :obl-litnbs-list { }
```

A.9 lpo2

```
declare operators
    lpo2 : Term Term --> Bool
    alpha2 : Termlist Term --> Bool
    beta2 : Term Term --> Bool
    gamma2 : Term Term --> Bool
    delta2 : Term Term --> Bool
    majo2 : Term Termlist --> Bool
    lex2 : Termlist Termlist --> Bool
    .
assert
  lex2-1 :
```

```
    lex2(nil,nil) = false
  lex2-2 :
    lex2(cons(t,ts),cons(u,us)) = lex2(ts,us)
      if t = u
  lex2-3 :
    lex2(cons(t,ts),cons(u,us)) = lpo2(t,u)
      if t =/= u
  majo2-1 :
    majo2(t,nil) = true
  majo2-2 :
    majo2(t,cons(u,us)) = and(lpo2(t,u),majo2(t,us))
  delta2-1 :
    delta2(F(f,ts),V(y)) = contains_tl(ts,y)
  gamma2-1 :
    gamma2(F(f,ts),F(g,us)) = and(lex2(ts,us),majo2(F(f,ts),us))
      if f = g
  gamma2-2 :
    gamma2(F(f,ts),F(g,us)) = false
      if f =/= g
  beta2-1 :
    beta2(F(f,ts),F(g,us)) = and(prec(f,g),majo2(F(f,ts),us))
  alpha2-1 :
    alpha2(nil,u) = false
  alpha2-2 :
    alpha2(cons(t,ts),u) = true
      if t = u
  alpha2-3 :
    alpha2(cons(t,ts),u) = or(lpo2(t,u),alpha2(ts,u))
      if t =/= u
  lpo2-1 :
    lpo2(V(x),u) = false
  lpo2-2 :
    lpo2(F(f,ts),V(y)) = delta2(F(f,ts),V(y))
  lpo2-3 :
    lpo2(F(f,ts),F(g,us)) = or(beta2(F(f,ts),F(g,us)),or(gamma2(F(f,ts),F(g,us)),
                                                         alpha2(ts,F(g,us))))
    .
call analyze-operator lpo2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator alpha2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator beta2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator gamma2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator delta2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator majo2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator lex2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lpo2(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo2-def-manuell
assume
    { def alpha2(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha2-def-manuell
assume
    { def beta2(t,u),
```

```
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    beta2-def-manuell
assume
    { def gamma2(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    gamma2-def-manuell
assume
    { def delta2(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta2-def-manuell
assume
    { def majo2(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo2-def-manuell
assume
    { def lex2(ts,us),
      length(ts) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true  }
    lex2-def-manuell
set weight (t,u,0) lpo2-def-manuell
set weight (ts,t) alpha2-def-manuell
set weight (t,u) beta2-def-manuell
set weight (t,u) gamma2-def-manuell
set weight (ts,us) lex2-def-manuell
set weight (t,us) majo2-def-manuell
call operators-strategy { 1 } { [1] } delta2-def-manuell
call activate-lemma  delta2-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { alpha2-def-manuell
         beta2-def-manuell gamma2-def-manuell } lpo2-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo2-def-manuell
         alpha2-def-manuell } alpha2-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { majo2-def-manuell
         beta2-def-manuell } beta2-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { majo2-def-manuell
         lex2-def-manuell gamma2-def-manuell } gamma2-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo2-def-manuell
         majo2-def-manuell } majo2-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo2-def-manuell
         lex2-def-manuell } lex2-def-manuell
call activate-lemmas { lpo2-def-manuell
                       alpha2-def-manuell
                       beta2-def-manuell
                       gamma2-def-manuell
                       majo2-def-manuell
                       lex2-def-manuell }
```

```
assume
    { lpo2(t,u) = lpo1(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo2-lpo1
assume
    { alpha2(ts,t) = alpha1(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha2-alpha1
assume
    { beta2(t,u) = beta1(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    beta2-beta1
assume
    { gamma2(t,u) = gamma1(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    gamma2-gamma1
assume
    { delta2(t,u) = delta1(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta2-delta1
assume
    { majo2(t,us) = majo1(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo2-majo1
assume
    { lex2(ts,us) = lex1(ts,us),
      length(ts) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lex2-lex1
set weight (t,u,0) lpo2-lpo1
set weight (ts,t) alpha2-alpha1
set weight (t,u) beta2-beta1
set weight (t,u) gamma2-gamma1
set weight (ts,us) lex2-lex1
set weight (t,us) majo2-majo1
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE delta2-delta1
call activate-lemma  delta2-delta1 :obl-litnbs-list { }
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { alpha2-alpha1 beta2-beta1 gamma2-gamma1 } lpo2-lpo1
call operators-strategy { 1 1 } { [1] [2] } ..
```

```
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { alpha2-alpha1 lpo2-lpo1 } alpha2-alpha1
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { beta2-beta1 majo2-majo1 } beta2-beta1
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { majo2-majo1 lex2-lex1 gamma2-gamma1 } gamma2-gamma1
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { majo2-majo1 lpo2-lpo1 } majo2-majo1
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { lpo2-lpo1 lex2-lex1 } lex2-lex1
call activate-lemma  lpo2-lpo1 :obl-litnbs-list { }
call activate-lemma  alpha2-alpha1 :obl-litnbs-list { }
call activate-lemma  beta2-beta1 :obl-litnbs-list { }
call activate-lemma  gamma2-gamma1 :obl-litnbs-list { }
call activate-lemma  majo2-majo1 :obl-litnbs-list { }
call activate-lemma  lex2-lex1 :obl-litnbs-list { }

declare operators
    lexM2 : Term Termlist Termlist --> Bool
    .
assert lexM2-1 :
    lexM2(t,nil,nil) = false
    .
assert lexM2-2 :
    lexM2(v,cons(t,ts),cons(u,us)) = lexM2(v,ts,us)
      if t = u
    .
assert lexM2-3 :
    lexM2(v,cons(t,ts),cons(u,us)) = and(lpo2(t,u),majo2(v,us))
      if t =/= u
    .
call analyze-operator lexM2 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lexM2(v,ts,us),
      Well(v) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true,
      length(ts) =/= length(us)}
    lexM2-def-manuell
call auto-strategy  lexM2-def-manuell
call activate-lemma  lexM2-def-manuell

assume
    { lexM2(F(f,ts1),ts,us) = and(Lex(ts,us),Majo(F(f,ts1),us)),
     sublist(ts,ts1) =/= true,
     arity(f) =/= length(ts1),
     Well_tl(ts1) =/= true,
     Well_tl(ts) =/= true,
     Well_tl(us) =/= true,
     length(ts) =/= length(us) }
    lexM2-lex2-majo2-h
call activate-lemma  Lpo-sublist
```

```
call auto-strategy  lexM2-lex2-majo2-h
call activate-lemma  lexM2-lex2-majo2-h
call deactivate-lemmas { Lpo-sublist }

assume
    { lexM2( F(f,ts),ts,us) = and(Lex(ts,us),Majo( F(f,ts),us)),
      length(ts) =/= length(us),
      arity(f) =/= length(ts),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexM2-lex2-majo2
call auto-strategy  lexM2-lex2-majo2
call deactivate-lemmas  { lexM2-lex2-majo2-h }
call activate-lemma  lexM2-lex2-majo2
```

A.10 lpo3

```
declare operators
    lpo3 : Term Term --> Bool
    alpha3 : Termlist Term --> Bool
    beta3 : Term Term --> Bool
    gamma3 : Term Term --> Bool
    delta3 : Term Term --> Bool
    majo3 : Term Termlist --> Bool
    lexM3 : Term Termlist Termlist --> Bool
    .

assert
  lexM3-1 :
    lexM3(v,nil,nil) = false
  lexM3-2 :
    lexM3(v,cons(t,ts),cons(u,us)) = lexM3(v,ts,us)
      if t = u
  lexM3-3 :
    lexM3(v,cons(t,ts),cons(u,us)) = and(lpo3(t,u),majo3(v,us))
      if t =/= u
  majo3-1 :
    majo3(t,nil) = true
  majo3-2 :
    majo3(t,cons(u,us)) = and(lpo3(t,u),majo3(t,us))
  delta3-1 :
    delta3(F(f,ts),V(y)) = contains_tl(ts,y)
  gamma3-1 :
    gamma3(F(f,ts),F(g,us)) = lexM3(F(f,ts),ts,us)
      if f = g
  gamma3-2 :
    gamma3(F(f,ts),F(g,us)) = false
      if f =/= g
  beta3-1 :
    beta3(F(f,ts),F(g,us)) = and(prec(f,g),majo3(F(f,ts),us))
  alpha3-1 :
    alpha3(nil,u) = false
  alpha3-2 :
    alpha3(cons(t,ts),u) = true
```

```
      if t = u
  alpha3-3 :
    alpha3(cons(t,ts),u) = or(lpo3(t,u),alpha3(ts,u))
      if t =/= u
  lpo3-1 :
    lpo3(V(x),u) = false
  lpo3-2 :
    lpo3(F(f,ts),V(y)) = delta3(F(f,ts),V(y))
  lpo3-3 :
    lpo3(F(f,ts),F(g,us)) = or(beta3(F(f,ts),F(g,us)),or(gamma3(F(f,ts),F(g,us)),
                                                         alpha3(ts,F(g,us))))
    .
call analyze-operator lpo3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator alpha3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator beta3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator gamma3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator delta3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator majo3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator lexM3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lpo3(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo3-def-manuell
assume
    { def alpha3(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha3-def-manuell
assume
    { def beta3(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    beta3-def-manuell
assume
    { def gamma3(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
    gamma3-def-manuell
assume
    { def delta3(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta3-def-manuell
assume
    { def majo3(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo3-def-manuell
assume
```

```
    { def lexM3(v,ts,us),
      Well(v) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true,
      length(ts) =/= length(us)}
    lexM3-def-manuell
set weight (u,t,0) lpo3-def-manuell
set weight (t,ts) alpha3-def-manuell
set weight (u,t) beta3-def-manuell
set weight (u,t) gamma3-def-manuell
set weight (us) lexM3-def-manuell
set weight (us,t) majo3-def-manuell
call operators-strategy { 1 } { [1] } delta3-def-manuell
call activate-lemma  delta3-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { alpha3-def-manuell
         beta3-def-manuell gamma3-def-manuell } lpo3-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo3-def-manuell
         alpha3-def-manuell } alpha3-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { majo3-def-manuell
         beta3-def-manuell } beta3-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lexM3-def-manuell
         gamma3-def-manuell } gamma3-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo3-def-manuell
         majo3-def-manuell } majo3-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo3-def-manuell
         majo3-def-manuell lexM3-def-manuell } lexM3-def-manuell
call activate-lemmas { lpo3-def-manuell
                       alpha3-def-manuell
                       beta3-def-manuell
                       gamma3-def-manuell
                       majo3-def-manuell
                       lexM3-def-manuell }

assume
    { lpo3(t,u) = lpo2(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo3-lpo2
assume
    { alpha3(ts,t) = alpha2(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha3-alpha2
assume
    { beta3(t,u) = beta2(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true   }
    beta3-beta2
assume
    { gamma3(t,u) = gamma2(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Fun(u) =/= true }
```

```
    gamma3-gamma2
assume
    { delta3(t,u) = delta2(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta3-delta2
assume
    { majo3(t,us) = majo2(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo3-majo2
assume
    { lexM3(v,ts,us) = lexM2(v,ts,us),
      length(ts) =/= length(us),
      Well(v) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexM3-lexM2
set weight (u,t,0) lpo3-lpo2
set weight (t,ts) alpha3-alpha2
set weight (u,t) beta3-beta2
set weight (u,t) gamma3-gamma2
set weight (us) lexM3-lexM2
set weight (us,t) majo3-majo2
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE delta3-delta2
call activate-lemma  delta3-delta2 :obl-litnbs-list { }
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { alpha3-alpha2 beta3-beta2 gamma3-gamma2 } lpo3-lpo2
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { alpha3-alpha2 lpo3-lpo2 } alpha3-alpha2
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { beta3-beta2 majo3-majo2 } beta3-beta2
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { majo3-majo2 lexM3-lexM2 gamma3-gamma2 } gamma3-gamma2
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { majo3-majo2 lpo3-lpo2 } majo3-majo2
call operators-strategy { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { lpo3-lpo2 majo3-majo2 lexM3-lexM2 } lexM3-lexM2
call activate-lemma  lpo3-lpo2 :obl-litnbs-list { }
call activate-lemma  alpha3-alpha2 :obl-litnbs-list { }
call activate-lemma  beta3-beta2 :obl-litnbs-list { }
call activate-lemma  gamma3-gamma2 :obl-litnbs-list { }
call activate-lemma  delta3-delta2 :obl-litnbs-list { }
call activate-lemma  majo3-majo2 :obl-litnbs-list { }
call activate-lemma  lexM3-lexM2 :obl-litnbs-list { }

declare operators
```

```
    lexMA3 : Term Term Termlist Termlist --> Bool
    .
assert
  lexMA3-1 :
    lexMA3(v,w,nil,nil) = false
  lexMA3-2 :
    lexMA3(v,w,cons(t,ts),cons(u,us)) = lexMA3(v,w,ts,us)
      if t = u
  lexMA3-3 :
    lexMA3(v,w,cons(t,ts),cons(u,us)) = majo3(v,us)
      if t =/= u,
         lpo3(t,u) = true
  lexMA3-4 :
    lexMA3(v,w,cons(t,ts),cons(u,us)) = alpha3(ts,w)
      if t =/= u,
         lpo3(t,u) =/= true,
         def lpo3(t,u)
    .
call analyze-operator lexMA3 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lexMA3(v,w,ts,us),
      Well(v) =/= true,
      Well(w) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true,
      length(ts) =/= length(us)}
    lexMA3-def-manuell
call auto-strategy  lexMA3-def-manuell
call activate-lemma  lexMA3-def-manuell

assume
    { lexMA3(F(f,ts1),F(g,us1),ts,us) = or(Alpha(ts,F(g,us1)),and(Lex(ts,us),
                                                           Majo(F(f,ts1),us))),
      sublist(ts,ts1) =/= true,
      sublist(us,us1) =/= true,
      length(ts) =/= length(us),
      arity(f) =/= length(ts1),
      arity(g) =/= length(us1),
      Well_tl(ts1) =/= true,
      Well_tl(us1) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMA3-lexM3-majo3-h
call deactivate-axioms  { Alpha-2 Alpha-3 Alpha-4 }
call operators-strategy  { 1 } { [1] } lexMA3-lexM3-majo3-h
call activate-axioms  { Alpha-4 Alpha-3 Alpha-2 Alpha-1 }
call activate-lemma  lexMA3-lexM3-majo3-h

assume
    { lexMA3(F(f,ts),F(g,us),ts,us) = or(alpha3(ts,F(g,us)),lexM3(F(f,ts),ts,us)),
      length(ts) =/= length(us),
      arity(f) =/= length(ts),
      arity(g) =/= length(us),
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMA3-lexM3-majo3
```

```
call simplify  lexMA3-lexM3-majo3
call activate-lemma lexMA3-lexM3-majo3
call deactivate-lemmas { lexMA3-lexM3-majo3-h }
```

A.11 lpo4

```
declare operators
    lpo4 : Term Term --> Bool
    alpha4 : Termlist Term --> Bool
    delta4 : Term Term --> Bool
    majo4 : Term Termlist --> Bool
    lexMA4 : Term Term Termlist Termlist --> Bool
    .
assert
  lexMA4-1 :
    lexMA4(v,w,nil,nil) = false
  lexMA4-2 :
    lexMA4(v,w,cons(t,ts),cons(u,us)) = lexMA4(v,w,ts,us)
      if t = u
  lexMA4-3 :
    lexMA4(v,w,cons(t,ts),cons(u,us)) = majo4(v,us)
      if t =/= u,
         lpo4(t,u) = true
  lexMA4-4 :
    lexMA4(v,w,cons(t,ts),cons(u,us)) = alpha4(ts,w)
      if t =/= u,
         lpo4(t,u) =/= true,
         def lpo4(t,u)
  majo4-1 :
    majo4(t,nil) = true
  majo4-2 :
    majo4(t,cons(u,us)) = and(lpo4(t,u),majo4(t,us))
  delta4-1 :
    delta4(F(f,ts),V(y)) = contains_tl(ts,y)
  alpha4-1 :
    alpha4(nil,u) = false
  alpha4-2 :
    alpha4(cons(t,ts),u) = true
      if t = u
  alpha4-3 :
    alpha4(cons(t,ts),u) = or(lpo4(t,u),alpha4(ts,u))
      if t =/= u
  lpo4-1 :
    lpo4(V(x),u) = false
  lpo4-2 :
    lpo4(F(f,ts),V(y)) = delta4(F(f,ts),V(y))
  lpo4-3 :
    lpo4(F(f,ts),F(g,us)) = majo4(F(f,ts),us)
      if prec(f,g) = true
  lpo4-4 :
    lpo4(F(f,ts),F(g,us)) = lexMA4(F(f,ts),F(g,us),ts,us)
      if prec(f,g) =/= true,
         def prec(f,g),
         f = g
```

```
  lpo4-5 :
    lpo4(F(f,ts),F(g,us)) = alpha4(ts,F(g,us))
      if prec(f,g) =/= true,
         def prec(f,g),
         f =/= g
    .
call analyze-operator lpo4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator alpha4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator delta4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator majo4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator lexMA4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lpo4(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo4-def-manuell
assume
    { def alpha4(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha4-def-manuell
assume
    { def delta4(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta4-def-manuell
assume
    { def majo4(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majo4-def-manuell
assume
    { def lexMA4(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMA4-def-manuell
set weight (t,u,t,u,0) lpo4-def-manuell
set weight (ts,t,ts,t) alpha4-def-manuell
set weight (t,us,t,us) majo4-def-manuell
set weight (F(f,vs),F(g,ws),ts,us) lexMA4-def-manuell
call operators-strategy { 1 } { [1] } delta4-def-manuell
call activate-lemma  delta4-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { alpha4-def-manuell
         majo4-def-manuell lexMA4-def-manuell } lpo4-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo4-def-manuell
         alpha4-def-manuell } alpha4-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo4-def-manuell
```

```
         majo4-def-manuell } majo4-def-manuell
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 } { [1] } :ind-lemmas { lpo4-def-manuell
         majo4-def-manuell alpha4-def-manuell lexMA4-def-manuell } lexMA4-def-manuell
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call activate-lemmas { lpo4-def-manuell
                       alpha4-def-manuell
                       majo4-def-manuell
                       lexMA4-def-manuell }

assume
    { lpo4(t,u) = lpo3(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo4-lpo3
assume
    { alpha4(ts,t) = alpha3(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alpha4-alpha3
assume
    { delta4(t,u) = delta3(t,u),
      Well(t) =/= true,
      Well(u) =/= true,
      Fun(t) =/= true,
      Var(u) =/= true   }
    delta4-delta3
assume
    { majo4(t,us) = majo3(t,us),
      Well_tl(us) =/= true,
      Well(t) =/= true }
    majo4-majo3
assume
    { lexMA4(F(f,vs),F(g,ws),ts,us) = lexMA3(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMA4-lexMA3
set weight (t,u,t,u,0) lpo4-lpo3
set weight (ts,t,ts,t) alpha4-alpha3
set weight (t,us,t,us) majo4-majo3
set weight (F(f,vs),F(g,ws),ts,us) lexMA4-lexMA3
call operators-strategy  { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE delta4-delta3
call activate-lemma  delta4-delta3 :obl-litnbs-list { }
call operators-strategy  { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { alpha4-alpha3 lexMA4-lexMA3 majo4-majo3 } lpo4-lpo3
call operators-strategy  { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
```

```
       :ind-lemmas { alpha4-alpha3 lpo4-lpo3 } alpha4-alpha3
call operators-strategy  { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { majo4-majo3 lpo4-lpo3 } majo4-majo3
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy  { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE ..
       :ind-lemmas { lpo4-lpo3 majo4-majo3  alpha4-alpha3 lexMA4-lexMA3 } lexMA4-lexMA3
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call activate-lemma  lpo4-lpo3 :obl-litnbs-list { }
call activate-lemma  alpha4-alpha3 :obl-litnbs-list { }
call activate-lemma  delta4-delta3 :obl-litnbs-list { }
call activate-lemma  majo4-majo3 :obl-litnbs-list { }
call activate-lemma  lexMA4-lexMA3 :obl-litnbs-list { }
```

A.12 lpoR4

```
declare operators
    lpoR4 : Term Term --> Res
    .
assert
  lpoR4-1 :
    lpoR4(t,u) = E
      if t = u
  lpoR4-2 :
    lpoR4(t,u) = G
      if t =/= u,
         lpo4(t,u) = true
  lpoR4-3 :
    lpoR4(t,u) = N
      if t =/= u,
         lpo4(t,u) =/= true,
         def lpo4(t,u)
    .
call analyze-operator lpoR4 :auto-insert-axioms-p FALSE ..
                            :speculate-domain-lemma-p FALSE
assume
    { def lpoR4(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpoR4-def-manuell
call operators-strategy  { 1 } { [1] } lpoR4-def-manuell
call activate-lemma  lpoR4-def-manuell

declare operators
    alphaR4 : Termlist Term --> Res
    .
assert
  alphaR4-1 :
    alphaR4(ts,u) = G
      if alpha4(ts,u) = true
  alphaR4-2 :
    alphaR4(ts,u) = N
      if alpha4(ts,u) =/= true,
```

```
        def alpha4(ts,u)
    .
call analyze-operator alphaR4 :auto-insert-axioms-p FALSE ..
                              :speculate-domain-lemma-p FALSE
assume
    { def alphaR4(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alphaR4-def-manuell
call operators-strategy  { 1 } { [1] } alphaR4-def-manuell
call activate-lemma  alphaR4-def-manuell

declare operators
    majoR4 : Term Termlist --> Res
    .
assert
  majoR4-1 :
    majoR4(t,us) = G
      if majo4(t,us) = true
  majoR4-2 :
    majoR4(t,us) = N
      if majo4(t,us) =/= true,
         def majo4(t,us)
    .
call analyze-operator majoR4 :auto-insert-axioms-p FALSE ..
                             :speculate-domain-lemma-p FALSE
assume
    { def majoR4(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majoR4-def-manuell
call operators-strategy  { 1 } { [1] } majoR4-def-manuell
call activate-lemma  majoR4-def-manuell

declare operators
    lexMAE4 : Term Term Termlist Termlist --> Res
    .
assert
  lexMAE4-1 :
    lexMAE4(v,w,ts,us) = E
      if ts = us
  lexMAE4-2 :
    lexMAE4(v,w,ts,us) = G
      if ts =/= us,
         lexMA4(v,w,ts,us) = true
  lexMAE4-3 :
    lexMAE4(v,w,ts,us) = N
      if ts =/= us,
         def lexMA4(v,w,ts,us),
         lexMA4(v,w,ts,us) =/= true
    .
call analyze-operator lexMAE4 :auto-insert-axioms-p FALSE ..
                              :speculate-domain-lemma-p FALSE
assume
    { def lexMAE4(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
```

```
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE4-def-manuell
call operators-strategy  { 1 } { [1] } lexMAE4-def-manuell
call activate-lemma  lexMAE4-def-manuell

assume
    { lpoR4(t,u) = E,
      lpoR4(t,u) = G,
      lpoR4(t,u) = N,
      Well(t) =/= true,
      Well(u) =/= true  }
    lpoR4-welldefined
call operators-strategy { 1 2 3 } { [1] [1] [1] } lpoR4-welldefined
call activate-lemma lpoR4-welldefined  :head-litnbs { 1 2 3 } :obl-litnbs-list {  }

assume
    { alphaR4(ts,t) = G,
      alphaR4(ts,t) = N,
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alphaR4-welldefined
call operators-strategy { 1 2 } { [1] [1] } alphaR4-welldefined

assume
    { majoR4(t,us) = G,
      majoR4(t,us) = N,
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majoR4-welldefined
call operators-strategy { 1 2 } { [1] [1] } majoR4-welldefined

assume
    { lexMAE4(F(f,vs),F(g,ws),ts,us) = E,
      lexMAE4(F(f,vs),F(g,ws),ts,us) = G,
      lexMAE4(F(f,vs),F(g,ws),ts,us) = N,
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE4-welldefined
call operators-strategy { 1 2 3 } { [1] [1] [1] } lexMAE4-welldefined

call activate-axiom  lpoR4-3 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 }
call activate-axiom  lpoR4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 }
```

```
call activate-axiom  lpoR4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
call activate-axiom  alphaR4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 }
call activate-axiom  alphaR4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
call activate-axiom  majoR4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 }
call activate-axiom  majoR4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
call activate-axiom  lexMAE4-3 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 }
call activate-axiom  lexMAE4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 }
call activate-axiom  lexMAE4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
```

A.13 lpoR5

```
declare operators
    lpoR5 : Term Term --> Res
    alphaR5 : Termlist Term --> Res
    majoR5 : Term Termlist --> Res
    lexMAE5 : Term Term Termlist Termlist --> Res
    .
assert
  lexMAE5-1 :
    lexMAE5(v,w,nil,nil) = E
  lexMAE5-2 :
    lexMAE5(v,w,cons(t,ts),cons(u,us)) = lexMAE5(v,w,ts,us)
      if lpoR5(t,u) = E
  lexMAE5-3 :
    lexMAE5(v,w,cons(t,ts),cons(u,us)) = majoR5(v,us)
      if lpoR5(t,u) = G
  lexMAE5-4 :
    lexMAE5(v,w,cons(t,ts),cons(u,us)) = alphaR5(ts,w)
      if lpoR5(t,u) = N
  majoR5-1 :
    majoR5(t,nil) = G
  majoR5-2 :
    majoR5(t,cons(u,us)) = majoR5(t,us)
      if lpoR5(t,u) = G
  majoR5-3 :
    majoR5(t,cons(u,us)) = N
      if lpoR5(t,u) = E
  majoR5-4 :
    majoR5(t,cons(u,us)) = N
      if lpoR5(t,u) = N
  alphaR5-1 :
    alphaR5(nil,u) = N
  alphaR5-2 :
    alphaR5(cons(t,ts),u) = G
      if lpoR5(t,u) = E
  alphaR5-3 :
    alphaR5(cons(t,ts),u) = G
      if lpoR5(t,u) = G
  alphaR5-4 :
    alphaR5(cons(t,ts),u) = alphaR5(ts,u)
      if lpoR5(t,u) = N
  lpoR5-1 :
    lpoR5(V(x),V(y)) = E
      if x = y
```

```
  lpoR5-2 :
    lpoR5(V(x),V(y)) = N
      if x =/= y
  lpoR5-3 :
    lpoR5(F(f,ts),V(y)) = G
      if contains_tl(ts,y) = true
  lpoR5-4 :
    lpoR5(F(f,ts),V(y)) = N
      if contains_tl(ts,y) =/= true,
         def contains_tl(ts,y)
  lpoR5-5 :
    lpoR5(V(x),F(g,us)) = N
  lpoR5-6 :
    lpoR5(F(f,ts),F(g,us)) = majoR5(F(f,ts),us)
      if prec(f,g) = true
  lpoR5-7 :
    lpoR5(F(f,ts),F(g,us)) = lexMAE5(F(f,ts),F(g,us),ts,us)
      if f = g,
         prec(f,g) =/= true,
         def prec(f,g)
  lpoR5-8 :
    lpoR5(F(f,ts),F(g,us)) = alphaR5(ts,F(g,us))
      if prec(f,g) =/= true,
         f =/= g,
         def prec(f,g)
    .
call analyze-operator lpoR5 :auto-insert-axioms-p FALSE ..
                            :speculate-domain-lemma-p FALSE
call analyze-operator alphaR5 :auto-insert-axioms-p FALSE ..
                              :speculate-domain-lemma-p FALSE
call analyze-operator majoR5 :auto-insert-axioms-p FALSE ..
                             :speculate-domain-lemma-p FALSE
call analyze-operator lexMAE5 :auto-insert-axioms-p FALSE ..
                              :speculate-domain-lemma-p FALSE
assume
    { def lpoR5(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpoR5-def-manuell
assume
    { def alphaR5(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alphaR5-def-manuell
assume
    { def majoR5(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majoR5-def-manuell
assume
    { def lexMAE5(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
```

```
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE5-def-manuell
assume
    { lpoR5(t,u) = E,
      lpoR5(t,u) = G,
      lpoR5(t,u) = N,
      Well(t) =/= true,
      Well(u) =/= true  }
    lpoR5-welldefined
assume
    { alphaR5(ts,t) = G,
      alphaR5(ts,t) = N,
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alphaR5-welldefined
assume
    { majoR5(t,us) = G,
      majoR5(t,us) = N,
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majoR5-welldefined
assume
    { lexMAE5(F(f,vs),F(g,ws),ts,us) = E,
      lexMAE5(F(f,vs),F(g,ws),ts,us) = G,
      lexMAE5(F(f,vs),F(g,ws),ts,us) = N,
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE5-welldefined
set weight (t,u,t) lpoR5-def-manuell
set weight (ts,t)  alphaR5-def-manuell
set weight (t,us)  majoR5-def-manuell
set weight (F(f,vs),F(g,ws),ts) lexMAE5-def-manuell
set weight (t,u,t) lpoR5-welldefined
set weight (ts,t)  alphaR5-welldefined
set weight (t,us)  majoR5-welldefined
set weight (F(f,vs),F(g,ws),ts) lexMAE5-welldefined
call operators-strategy { 1 } { [1] } :ind-lemmas { alphaR5-def-manuell
         majoR5-def-manuell lexMAE5-def-manuell } lpoR5-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpoR5-def-manuell
         alphaR5-def-manuell lpoR5-welldefined } alphaR5-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpoR5-def-manuell
         majoR5-def-manuell lpoR5-welldefined } majoR5-def-manuell
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 } { [1] } :ind-lemmas { lpoR5-def-manuell
         majoR5-def-manuell alphaR5-def-manuell lexMAE5-def-manuell
         lpoR5-welldefined } lexMAE5-def-manuell
```

```
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 2 3 } { [1] [1] [1] } :ind-lemmas { alphaR5-def-manuell
         majoR5-def-manuell lexMAE5-def-manuell majoR5-welldefined
         lexMAE5-welldefined alphaR5-welldefined} lpoR5-welldefined
call operators-strategy { 1 2 } { [1] [1] } :ind-lemmas { lpoR5-def-manuell
         lpoR5-welldefined alphaR5-welldefined} alphaR5-welldefined
call operators-strategy { 1 2 } { [1] [1] } :ind-lemmas { lpoR5-def-manuell
         lpoR5-welldefined majoR5-welldefined} majoR5-welldefined
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 2 3 } { [1] [1] [1] } :ind-lemmas { lpoR5-def-manuell
         majoR5-def-manuell alphaR5-def-manuell lexMAE5-def-manuell
         lpoR5-welldefined lexMAE5-welldefined majoR5-welldefined
         alphaR5-welldefined} lexMAE5-welldefined
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call activate-lemmas { lpoR5-def-manuell alphaR5-def-manuell majoR5-def-manuell
                       lexMAE5-def-manuell }

assume
    { lpoR5(t,u) = lpoR4(t,u),
      Well(t) =/= true,
      Well(u) =/= true }
    lpoR5-lpoR4
assume
    { lexMAE5(F(f,vs),F(g,ws),ts,us) = lexMAE4(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE5-lexMAE4
assume
    { majoR5(t,us) = majoR4(t,us),
      Well_tl(us) =/= true,
      Well(t) =/= true  }
    majoR5-majoR4
assume
    { alphaR5(ts,u) = alphaR4(ts,u),
      Well_tl(ts) =/= true,
      Well(u) =/= true  }
    alphaR5-alphaR4
set weight (t,u,t) lpoR5-lpoR4
set weight (ts,u)  alphaR5-alphaR4
set weight (t,us)  majoR5-majoR4
set weight (F(f,vs),F(g,ws),ts) lexMAE5-lexMAE4
call activate-lemma  Lemma3-consequence-4 :free-vars-bindings { { "lit(2)" } }
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { lpoR5-lpoR4
         lexMAE5-lexMAE4 majoR5-majoR4 alphaR5-alphaR4 } ..
       :allow-simplification-before-induction-p FALSE lpoR5-lpoR4
call activate-lemma  lexMA3-lexM3-majo3-h
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call deactivate-axioms  { Lpo-1 Lpo-2 Lpo-3 Lpo-4 Lpo-5 }
call deactivate-lemmas  { Lemma3-consequence-5 Lemma3-consequence-5a }
```

```
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { lpoR5-lpoR4
         lexMAE5-lexMAE4 majoR5-majoR4 alphaR5-alphaR4 } ..
       :allow-simplification-before-induction-p FALSE lexMAE5-lexMAE4
call activate-axioms  { Lpo-5 Lpo-4 Lpo-3 Lpo-2 Lpo-1 }
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { lpoR5-lpoR4
         lexMAE5-lexMAE4 majoR5-majoR4 alphaR5-alphaR4 } ..
       :allow-simplification-before-induction-p FALSE majoR5-majoR4
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { lpoR5-lpoR4
         lexMAE5-lexMAE4 majoR5-majoR4 alphaR5-alphaR4 } ..
       :allow-simplification-before-induction-p FALSE alphaR5-alphaR4
call activate-lemma  lpoR5-lpoR4 :obl-litnbs-list {  }
call activate-lemma  lexMAE5-lexMAE4 :obl-litnbs-list {  }
call activate-lemma  majoR5-majoR4 :obl-litnbs-list {  }
call activate-lemma  alphaR5-alphaR4 :obl-litnbs-list {  }

declare operators
    lpo5 : Term Term --> Bool
    .
assert
  lpo5-1 :
    lpo5(t,u) = true
      if lpoR5(t,u) = G
  lpo5-2 :
    lpo5(t,u) = false
      if lpoR5(t,u) = E
  lpo5-3 :
    lpo5(t,u) = false
      if lpoR5(t,u) = N
    .
call analyze-operator lpo5 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def lpo5(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpo5-def-manuell
call operators-strategy  { 1 } { [1] } lpo5-def-manuell
call activate-lemma lpo5-def-manuell

assume
    { lpo5(t,u) = lpo4(t,u),
      Well(t) =/= true,
      Well(u) =/= true }
    lpo5-lpo4
call activate-lemma  Alpha-irrefl
call operators-strategy  { 1 1 } { [1] [2] } ..
       :allow-simplification-before-induction-p FALSE lpo5-lpo4
call activate-lemma  Alpha-irrefl :obl-litnbs-list { { 2 } }
call activate-lemma  lpo5-lpo4 :obl-litnbs-list { }
```

A.14 lpoR6

```
declare operators
    lpoR6 : Term Term --> Res
```

```
    alphaR6 : Termlist Term --> Res
    majoR6 : Term Termlist --> Res
    lexMAE6 : Term Term Termlist Termlist --> Res
    .
assert
  lexMAE6-1 :
    lexMAE6(v,w,nil,nil) = E
  lexMAE6-2 :
    lexMAE6(v,w,cons(t,ts),cons(u,us)) = lexMAE6(v,w,ts,us)
      if lpoR6(t,u) = E
  lexMAE6-3 :
    lexMAE6(v,w,cons(t,ts),cons(u,us)) = majoR6(v,us)
      if lpoR6(t,u) = G
  lexMAE6-4 :
    lexMAE6(v,w,cons(t,ts),cons(u,us)) = alphaR6(ts,w)
      if lpoR6(t,u) = N
  majoR6-1 :
    majoR6(t,nil) = G
  majoR6-2 :
    majoR6(t,cons(u,us)) = majoR6(t,us)
      if lpoR6(t,u) = G
  majoR6-3 :
    majoR6(t,cons(u,us)) = N
      if lpoR6(t,u) = E
  majoR6-4 :
    majoR6(t,cons(u,us)) = N
      if lpoR6(t,u) = N
  alphaR6-1 :
    alphaR6(nil,u) = N
  alphaR6-2 :
    alphaR6(cons(t,ts),u) = G
      if lpoR6(t,u) = E
  alphaR6-3 :
    alphaR6(cons(t,ts),u) = G
      if lpoR6(t,u) = G
  alphaR6-4 :
    alphaR6(cons(t,ts),u) = alphaR6(ts,u)
      if lpoR6(t,u) = N
  lpoR6-1 :
    lpoR6(V(x),V(y)) = E
      if x = y
  lpoR6-2 :
    lpoR6(V(x),V(y)) = N
      if x =/= y
  lpoR6-3 :
    lpoR6(F(f,ts),V(y)) = G
      if contains_tl(ts,y) = true
  lpoR6-4 :
    lpoR6(F(f,ts),V(y)) = N
      if contains_tl(ts,y) =/= true,
         def contains_tl(ts,y)
  lpoR6-5 :
    lpoR6(V(x),F(g,us)) = N
  lpoR6-6 :
    lpoR6(F(f,ts),F(g,us)) = majoR6(F(f,ts),us)
      if prec(f,g) = true
```

```
  lpoR6-7 :
    lpoR6(F(f,ts),F(g,us)) = lexMAE6(F(f,ts),F(g,us),ts,us)
      if f = g,
         prec(f,g) =/= true,
         def prec(f,g)
  lpoR6-8 :
    lpoR6(F(f,ts),F(g,us)) = alphaR6(ts,F(g,us))
      if prec(f,g) =/= true,
         f =/= g,
         def prec(f,g)
    .
call analyze-operator lpoR6 :auto-insert-axioms-p FALSE ..
                            :speculate-domain-lemma-p FALSE
call analyze-operator alphaR6 :auto-insert-axioms-p FALSE ..
                              :speculate-domain-lemma-p FALSE
call analyze-operator majoR6 :auto-insert-axioms-p FALSE ..
                             :speculate-domain-lemma-p FALSE
call analyze-operator lexMAE6 :auto-insert-axioms-p FALSE ..
                              :speculate-domain-lemma-p FALSE
assume
    { def lpoR6(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    lpoR6-def-manuell
assume
    { def alphaR6(ts,t),
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alphaR6-def-manuell
assume
    { def majoR6(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majoR6-def-manuell
assume
    { def lexMAE6(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE6-def-manuell
assume
    { lpoR6(t,u) = E,
      lpoR6(t,u) = G,
      lpoR6(t,u) = N,
      Well(t) =/= true,
      Well(u) =/= true  }
    lpoR6-welldefined
assume
    { alphaR6(ts,t) = G,
      alphaR6(ts,t) = N,
```

```
      Well_tl(ts) =/= true,
      Well(t) =/= true  }
    alphaR6-welldefined
assume
    { majoR6(t,us) = G,
      majoR6(t,us) = N,
      Well(t) =/= true,
      Well_tl(us) =/= true }
    majoR6-welldefined
assume
    { lexMAE6(F(f,vs),F(g,ws),ts,us) = E,
      lexMAE6(F(f,vs),F(g,ws),ts,us) = G,
      lexMAE6(F(f,vs),F(g,ws),ts,us) = N,
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE6-welldefined
set weight (t,u,t) lpoR6-def-manuell
set weight (ts,t)  alphaR6-def-manuell
set weight (t,us)  majoR6-def-manuell
set weight (F(f,vs),F(g,ws),ts) lexMAE6-def-manuell
set weight (t,u,t) lpoR6-welldefined
set weight (ts,t)  alphaR6-welldefined
set weight (t,us)  majoR6-welldefined
set weight (F(f,vs),F(g,ws),ts) lexMAE6-welldefined
call operators-strategy { 1 } { [1] } :ind-lemmas { alphaR6-def-manuell
         majoR6-def-manuell lexMAE6-def-manuell } lpoR6-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpoR6-def-manuell
         alphaR6-def-manuell lpoR6-welldefined } alphaR6-def-manuell
call operators-strategy { 1 } { [1] } :ind-lemmas { lpoR6-def-manuell
         majoR6-def-manuell lpoR6-welldefined } majoR6-def-manuell
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 } { [1] } :ind-lemmas { lpoR6-def-manuell
         majoR6-def-manuell alphaR6-def-manuell lexMAE6-def-manuell
         lpoR6-welldefined } lexMAE6-def-manuell
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 2 3 } { [1] [1] [1] } :ind-lemmas { alphaR6-def-manuell
         majoR6-def-manuell lexMAE6-def-manuell majoR6-welldefined
         lexMAE6-welldefined alphaR6-welldefined} lpoR6-welldefined
call operators-strategy { 1 2 } { [1] [1] } :ind-lemmas { lpoR6-def-manuell
         lpoR6-welldefined alphaR6-welldefined} alphaR6-welldefined
call operators-strategy { 1 2 } { [1] [1] } :ind-lemmas { lpoR6-def-manuell
         lpoR6-welldefined majoR6-welldefined} majoR6-welldefined
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy { 1 2 3 } { [1] [1] [1] } :ind-lemmas { lpoR6-def-manuell
         majoR6-def-manuell alphaR6-def-manuell lexMAE6-def-manuell
         lpoR6-welldefined lexMAE6-welldefined majoR6-welldefined
         alphaR6-welldefined} lexMAE6-welldefined
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call activate-lemmas { lpoR6-def-manuell alphaR6-def-manuell majoR6-def-manuell
```

```
                    lexMAE6-def-manuell }

assume
    { lpoR6(t,u) = lpoR5(t,u),
      Well(t) =/= true,
      Well(u) =/= true }
    lpoR6-lpoR5
assume
    { alphaR6(ts,u) = alphaR5(ts,u),
      Well_tl(ts) =/= true,
      Well(u) =/= true }
    alphaR6-alphaR5
assume
    { majoR6(t,us) = majoR5(t,us),
      Well_tl(us) =/= true,
      Well(t) =/= true }
    majoR6-majoR5
assume
    { lexMAE6(F(f,vs),F(g,ws),ts,us) = lexMAE5(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    lexMAE6-lexMAE5
set weight (t,u,t) lpoR6-lpoR5
set weight (ts,u)  alphaR6-alphaR5
set weight (t,us)  majoR6-majoR5
set weight (F(f,vs),F(g,ws),ts) lexMAE6-lexMAE5
call operators-strategy { 1 1 } { [2] [1] } :ind-lemmas { lpoR6-lpoR5
         alphaR6-alphaR5 majoR6-majoR5 lexMAE6-lexMAE5 } ..
       :allow-simplification-before-induction-p FALSE lpoR6-lpoR5
call operators-strategy { 1 1 } { [2] [1] } :ind-lemmas { lpoR6-lpoR5
         alphaR6-alphaR5 majoR6-majoR5 lexMAE6-lexMAE5 } ..
       :allow-simplification-before-induction-p FALSE alphaR6-alphaR5
call operators-strategy { 1 1 } { [2] [1] } :ind-lemmas { lpoR6-lpoR5
         alphaR6-alphaR5 majoR6-majoR5 lexMAE6-lexMAE5 } ..
       :allow-simplification-before-induction-p FALSE majoR6-majoR5
call activate-lemmas  { F-sublist-ind subterm_tl-ind }
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { lpoR6-lpoR5
         alphaR6-alphaR5 majoR6-majoR5 lexMAE6-lexMAE5 } ..
       :allow-simplification-before-induction-p FALSE lexMAE6-lexMAE5
call deactivate-lemmas  { F-sublist-ind subterm_tl-ind }
call activate-lemma  lpoR6-lpoR5 :obl-litnbs-list { }
call activate-lemma  alphaR6-alphaR5 :obl-litnbs-list { }
call activate-lemma  majoR6-majoR5 :obl-litnbs-list { }
call activate-lemma  lexMAE6-lexMAE5 :obl-litnbs-list { }
```

A.15 clpo4

```
declare operators
    clpo4 : Term Term --> Res
    .
assert
  clpo4-1 :
    clpo4(t,u) = E
      if t = u
  clpo4-2 :
    clpo4(t,u) = G
      if t =/= u,
         lpo4(t,u) = true
  clpo4-3 :
    clpo4(t,u) = L
      if t =/= u,
         def lpo4(t,u),
         lpo4(t,u) =/= true,
         lpo4(u,t) = true
  clpo4-4 :
    clpo4(t,u) = N
      if t =/= u,
         def lpo4(t,u),
         lpo4(t,u) =/= true,
         def lpo4(u,t),
         lpo4(u,t) =/= true
    .
call analyze-operator clpo4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def clpo4(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    clpo4-def-manuell
call operators-strategy  { 1 } { [1] } clpo4-def-manuell
call activate-lemma  clpo4-def-manuell

declare operators
    cMA4 : Term Termlist --> Res
    .
assert
  cMA4-1 :
    cMA4(t,us) = G
      if majo4(t,us) = true
  cMA4-2 :
    cMA4(t,us) = L
      if def majo4(t,us),
         majo4(t,us) =/= true,
         alpha4(us,t) = true
  cMA4-3 :
    cMA4(t,us) = N
      if def majo4(t,us),
         majo4(t,us) =/= true,
         def alpha4(us,t),
         alpha4(us,t) =/= true
    .
call analyze-operator cMA4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
```

```
assume
    { def cMA4(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    cMA4-def-manuell
call operators-strategy  { 1 } { [1] } cMA4-def-manuell
call activate-lemma  cMA4-def-manuell

declare operators
    cLMA4 : Term Term Termlist Termlist --> Res
    .
assert
  cLMA4-1 :
    cLMA4(t,u,ts,us) = E
      if ts = us
  cLMA4-2 :
    cLMA4(t,u,ts,us) = G
      if ts =/= us,
         lexMA4(t,u,ts,us) = true
  cLMA4-3 :
    cLMA4(t,u,ts,us) = L
      if ts =/= us,
         def lexMA4(t,u,ts,us),
         lexMA4(t,u,ts,us) =/= true,
         lexMA4(u,t,us,ts) = true
  cLMA4-4 :
    cLMA4(t,u,ts,us) = N
      if ts =/= us,
         def lexMA4(t,u,ts,us),
         lexMA4(t,u,ts,us) =/= true,
         def lexMA4(u,t,us,ts),
         lexMA4(u,t,us,ts) =/= true
    .
call analyze-operator cLMA4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def cLMA4(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    cLMA4-def-manuell
call operators-strategy  { 1 } { [1] } cLMA4-def-manuell
call activate-lemma  cLMA4-def-manuell

declare operators
    cAA4 : Term Term Termlist Termlist --> Res
    .
assert
  cAA4-1 :
    cAA4(t,u,ts,us) = G
      if alpha4(ts,u) = true
```

```
  cAA4-2 :
    cAA4(t,u,ts,us) = L
      if def alpha4(ts,u),
         alpha4(ts,u) =/= true,
         alpha4(us,t) = true
  cAA4-3 :
    cAA4(t,u,ts,us) = N
      if def alpha4(ts,u),
         alpha4(ts,u) =/= true,
         def alpha4(us,t),
         alpha4(us,t) =/= true
    .
call analyze-operator cAA4 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def cAA4(t,u,ts,us),
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    cAA4-def-manuell
call operators-strategy  { 1 } { [1] } cAA4-def-manuell
call activate-lemma  cAA4-def-manuell

assume
    { cMA4(t,us) = G,
      cMA4(t,us) = L,
      cMA4(t,us) = N,
      Well(t) =/= true,
      Well_tl(us) =/= true}
    cMA4-welldefined
call operators-strategy  { 1 2 3 } { [1] [1] [1] } cMA4-welldefined

assume
    { cAA4(t,u,ts,us) = G,
      cAA4(t,u,ts,us) = L,
      cAA4(t,u,ts,us) = N,
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true}
    cAA4-welldefined
call operators-strategy  { 1 2 3 } { [1] [1] [1] } cAA4-welldefined

call activate-axiom  clpo4-4 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 5 6 }
call activate-axiom  clpo4-3 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 5 }
call activate-axiom  clpo4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 }
call activate-axiom  clpo4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
call activate-axiom  cMA4-3 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 5 }
call activate-axiom  cMA4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 }
call activate-axiom  cMA4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
call activate-axiom  cLMA4-4 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 5 6 }
call activate-axiom  cLMA4-3 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 5 }
call activate-axiom  cLMA4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 }
call activate-axiom  cLMA4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
call activate-axiom  cAA4-3 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 5 }
call activate-axiom  cAA4-2 :obl-litnbs-list {  } :generous-litnbs { 1 2 3 4 }
```

```
call activate-axiom  cAA4-1 :obl-litnbs-list {  } :generous-litnbs { 1 2 }
```

A.16 clpo5

```
declare operators
    clpo5 : Term Term --> Res
    .
assert
  clpo5-1 :
    clpo5(t,u) = E
      if lpoR5(t,u) = E
  clpo5-2 :
    clpo5(t,u) = G
      if lpoR5(t,u) = G
  clpo5-3 :
    clpo5(t,u) = flip(lpoR5(u,t))
      if lpoR5(t,u) = N
    .
call analyze-operator clpo5 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def clpo5(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    clpo5-def-manuell
call operators-strategy  { 1 } { [1] } clpo5-def-manuell
call activate-lemma  clpo5-def-manuell

assume
    { clpo5(t,u) = clpo4(t,u),
      Well(t) =/= true,
      Well(u) =/= true }
    clpo5-clpo4
call operators-strategy { 1 1 } { [2] [1] } ..
       :allow-simplification-before-induction-p FALSE clpo5-clpo4
call activate-lemma  clpo5-clpo4 :obl-litnbs-list { }

call deactivate-axioms  { clpo5-1 clpo5-2 clpo5-3 }
```

A.17 clpo6

```
declare operators
    clpo6 : Term Term --> Res
    cMA6 : Term Termlist --> Res
    cLMA6 : Term Term Termlist Termlist --> Res
    cAA6 : Term Term Termlist Termlist --> Res
    .
assert
  clpo6-1 :
    clpo6(F(f,ts),V(y)) = G
    if
      contains_tl(ts,y) = true
  clpo6-2 :
```

```
  clpo6(F(f,ts),V(y)) = N
    if def contains_tl(ts,y),
       contains_tl(ts,y) =/= true
clpo6-3 :
  clpo6(V(x),F(g,us)) = L
    if contains_tl(us,x) = true
clpo6-4 :
  clpo6(V(x),F(g,us)) = N
    if def contains_tl(us,x),
       contains_tl(us,x) =/= true
clpo6-5 :
  clpo6(V(x),V(y)) = E
    if x = y
clpo6-6 :
  clpo6(V(x),V(y)) = N
    if x =/= y
clpo6-7 :
  clpo6(F(f,ts),F(g,us)) = cLMA6(F(f,ts),F(g,us),ts,us)
    if f = g
clpo6-8 :
  clpo6(F(f,ts),F(g,us)) = cMA6(F(f,ts),us)
    if f =/= g,
       prec(f,g) = true
clpo6-9 :
  clpo6(F(f,ts),F(g,us)) = flip(cMA6(F(g,us),ts))
    if f =/= g,
       def prec(f,g),
       prec(f,g) =/= true,
       prec(g,f) = true
clpo6-10 :
  clpo6(F(f,ts),F(g,us)) = cAA6(F(f,ts),F(g,us),ts,us)
    if f =/= g,
       def prec(f,g),
       prec(f,g) =/= true,
       def prec(g,f),
       prec(g,f) =/= true
cMA6-1 :
  cMA6(t,nil) = G
cMA6-2 :
  cMA6(t,cons(u,us)) = cMA6(t,us)
    if clpo6(t,u) = G
cMA6-3 :
  cMA6(t,cons(u,us)) = L
    if clpo6(t,u) = E
cMA6-4 :
  cMA6(t,cons(u,us)) = L
    if clpo6(t,u) = L
cMA6-5 :
  cMA6(t,cons(u,us)) = flip(alphaR6(us,t))
    if clpo6(t,u) = N
cLMA6-1 :
  cLMA6(t,u,nil,nil) = E
cLMA6-2 :
  cLMA6(t,u,cons(ti,ts),cons(ui,us)) = cLMA6(t,u,ts,us)
    if clpo6(ti,ui) = E
cLMA6-3 :
```

```
    cLMA6(t,u,cons(ti,ts),cons(ui,us)) = cMA6(t,us)
      if clpo6(ti,ui) = G
  cLMA6-4 :
    cLMA6(t,u,cons(ti,ts),cons(ui,us)) = flip(cMA6(u,ts))
      if clpo6(ti,ui) = L
  cLMA6-5 :
    cLMA6(t,u,cons(ti,ts),cons(ui,us)) = cAA6(t,u,ts,us)
      if clpo6(ti,ui) = N
  cAA6-1 :
    cAA6(t,u,ts,us) = G
      if alphaR6(ts,u) = G
  cAA6-2 :
    cAA6(t,u,ts,us) = flip(alphaR6(us,t))
      if alphaR6(ts,u) = N
    .
call analyze-operator clpo6 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator cMA6 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator cLMA6 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
call analyze-operator cAA6 :auto-insert-axioms-p FALSE :speculate-domain-lemma-p FALSE
assume
    { def clpo6(t,u),
      Well(t) =/= true,
      Well(u) =/= true  }
    clpo6-def-manuell
assume
    { def cMA6(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    cMA6-def-manuell
assume
    { def cLMA6(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    cLMA6-def-manuell
assume
    { def cAA6(t,u,ts,us),
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    cAA6-def-manuell
assume
    { cMA6(t,us) = G,
      cMA6(t,us) = L,
      cMA6(t,us) = N,
      Well(t) =/= true,
      Well_tl(us) =/= true}
    cMA6-welldefined
assume
```

```
    { cAA6(t,u,ts,us) = G,
      cAA6(t,u,ts,us) = L,
      cAA6(t,u,ts,us) = N,
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true}
    cAA6-welldefined
set weight (+(term-size(t),term-size(u)),term-size(t)) clpo6-def-manuell
set weight (+(term-size(t),term-size_tl(us))) cMA6-def-manuell
set weight (+(term-size(F(f,vs)),term-size(F(g,ws))),term-size_tl(ts)) cLMA6-def-manuell
set weight (+(term-size(t),term-size_tl(us))) cMA6-welldefined
call operators-strategy  { 1 } { [1] } cAA6-def-manuell
call activate-lemma cAA6-def-manuell
call operators-strategy  { 1 2 3 } { [1] [1] [1] } cAA6-welldefined
call operators-strategy  { 1 } { [1] } :ind-lemmas { cMA6-def-manuell
         cLMA6-def-manuell } clpo6-def-manuell
call operators-strategy  { 1 } { [1] } :ind-lemmas { clpo6-def-manuell
         cMA6-def-manuell } cMA6-def-manuell
call operators-strategy  { 1 } { [1] } :ind-lemmas { clpo6-def-manuell
         cMA6-def-manuell cLMA6-def-manuell cMA6-welldefined } cLMA6-def-manuell
call operators-strategy  { 1 2 3 } { [1] [1] [1] } :ind-lemmas { clpo6-def-manuell
         cMA6-welldefined } cMA6-welldefined
call activate-lemmas { clpo6-def-manuell cMA6-def-manuell cLMA6-def-manuell }

assume
    { clpo6(t,u) = clpo4(t,u),
      Well(t) =/= true,
      Well(u) =/= true }
    clpo6-clpo4
assume
    { cMA6(t,us) = cMA4(t,us),
      Well(t) =/= true,
      Well_tl(us) =/= true }
    cMA6-cMA4
assume
    { cLMA6(F(f,vs),F(g,ws),ts,us) = cLMA4(F(f,vs),F(g,ws),ts,us),
      sublist(ts,vs) =/= true,
      sublist(us,ws) =/= true,
      arity(f) =/= length(vs),
      arity(g) =/= length(ws),
      length(ts) =/= length(us),
      Well_tl(vs) =/= true,
      Well_tl(ws) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    cLMA6-cLMA4
assume
    { cAA6(t,u,ts,us) = cAA4(t,u,ts,us),
      Well(t) =/= true,
      Well(u) =/= true,
      Well_tl(ts) =/= true,
      Well_tl(us) =/= true }
    cAA6-cAA4
set weight (+(term-size(t),term-size(u)),term-size(t)) clpo6-clpo4
set weight (+(term-size(t),term-size_tl(us))) cMA6-cMA4
```

```
set weight (+(term-size(F(f,vs)),term-size(F(g,ws))),term-size_tl(ts)) cLMA6-cLMA4
call operators-strategy  { 1 1 } { [2] [1] } ..
       :allow-simplification-before-induction-p FALSE cAA6-cAA4
call activate-lemma cAA6-cAA4 :obl-litnbs-list { }
call activate-lemma  Lpo-irrefl-trans
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { cMA6-cMA4
         cLMA6-cLMA4 } :allow-simplification-before-induction-p FALSE clpo6-clpo4
call activate-lemma  Lpo-irrefl-trans :head-litnbs { 1 } :obl-litnbs-list { { 2 } }
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { clpo6-clpo4
         cMA6-cMA4 } :allow-simplification-before-induction-p FALSE cMA6-cMA4
call deactivate-axioms  { Beta-1 Beta-2 Gamma-1 Gamma-2 Gamma-3 }
call operators-strategy  { 1 1 } { [2] [1] } :ind-lemmas { clpo6-clpo4
         cMA6-cMA4 cLMA6-cLMA4 } :allow-simplification-before-induction-p FALSE ..
       :rule-type-order { IND LMA AX } cLMA6-cLMA4
call activate-axioms  { Beta-2 Beta-1 Gamma-3 Gamma-2 Gamma-1 }
call activate-lemma clpo6-clpo4 :obl-litnbs-list { }
call activate-lemma cMA6-cMA4 :obl-litnbs-list { }
call activate-lemma cLMA6-cLMA4 :obl-litnbs-list { }
```

Bibliography

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1–2):133–178, 2000.

[AKSSW03] J. Avenhaus, U. Kühler, T. Schmidt-Samoa, and C.-P. Wirth. How to prove inductive theorems? QuodLibet! In Baader [Baa03], pages 328–333.

[AM90] J. Avenhaus and K. Madlener. Term rewriting and equational reasoning. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 1–43. North-Holland, 1990.

[AM97] J. Avenhaus and K. Madlener. Theorem proving in hierarchical causal specifications. In *Advances in Algorithms, Languages, and Complexity*, pages 1–51, 1997.

[AR01] A. Armando and S. Ranise. A practical extension mechanism for decision procedures: the case study of universal presburger arithmetic. *J. UCS*, 7(2):124–140, 2001.

[AR03] A. Armando and S. Ranise. Constraint contextual rewriting. *J. Symb. Comput.*, 36(1-2):193–216, 2003.

[ARS02] A. Armando, M. Rusinowitch, and S. Stratulat. Incorporating decision procedures in implicit induction. *J. Symb. Comput.*, 34(4):241–258, 2002.

[Ave95] J. Avenhaus. *Reduktionssysteme: Rechnen und Schließen in gleichungsdefinierten Strukturen.* Springer-Verlag, 1995.

[Baa03] F. Baader, editor. *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *LNCS*. Springer, 2003.

[Bac88] L. Bachmair. Proof by consistency in equational theories. In *LICS*, pages 228–233. IEEE Computer Society, 1988.

[BC96] A. Boudet and H. Comon. Diophantine equations, presburger arithmetic and finite automata. In H. Kirchner, editor, *CAAP*, volume 1059 of *LNCS*, pages 30–43. Springer, 1996.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[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, *TACAS*, volume 2619 of *LNCS*, pages 521–536. Springer, 2003.

[Ble75] W. W. Bledsoe. A new method for proving certain presburger formulas. In *IJCAI*, pages 15–21, 1975.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979.

[BM88a] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., 1988.

[BM88b] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., 1988.

[BM91] R. S. Boyer and J S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[BR93] A Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In *IJCAI*, pages 88–94, 1993.

[Bru91] M. Bruynooghe. Intelligent backtracking revisited. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 166–177, 1991.

[BS97] R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI/IAAI*, pages 203–208, 1997.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2):185–253, 1993.

[Bun01] A. Bundy. The automation of proof by mathematical induction. In Robinson and Voronkov [RV01], pages 845–911.

[Com01] H. Comon. Inductionless induction. In Robinson and Voronkov [RV01], pages 913–962.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. Edinburgh University Press, 1972.

[Den05] L. Dennis. Induction challenge problems. `http://www.cs.nott.ac.uk/~lad/research/challenges/index.html`, 2005.

[Der87] N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[Fre60] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[Gen35] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1934/35.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[GK03] J. Giesl and D. Kapur. Deciding inductive validity of equations. In Baader [Baa03], pages 17–31.

[GS92] H. Ganzinger and J. Stuber. Inductive theorem proving by consistency for first-order clauses. In M. Rusinowitch and J.-L. Remy, editors, *CTRS*, volume 656 of *LNCS*, pages 226–241. Springer, 1992.

[GTSKF04] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In V. van Oostrom, editor, *RTA*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

[HKM03] W. A. Hunt Jr., R. B. Krug, and J S. Moore. Linear and nonlinear arithmetic in ACL2. In D. Geist and E. Tronci, editors, *CHARME*, volume 2860 of *LNCS*, pages 319–333. Springer, 2003.

[HKM04] W. A. Hunt Jr., R. B. Krug, and J S. Moore. Integrating nonlinear arithmetic into ACL2. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004.

[Hod71] L. Hodes. Solving problems by formula manipulation in logic and linear inequalities. In *IJCAI*, pages 553–559, 1971.

[Hut97] D. Hutter. Coloring terms to control equational reasoning. *J. Autom. Reasoning*, 18(3):399–442, 1997.

[JB02] P. Janicic and A. Bundy. A general setting for flexibly combining and augmenting decision procedures. *J. Autom. Reasoning*, 28(3):257–305, 2002.

[JBG99] P. Janicic, A. Bundy, and I. Green. A framework for the flexible integration of a class of decision procedures into theorem provers. In H. Ganzinger, editor, *CADE*, volume 1632 of *LNCS*, pages 127–141. Springer, 1999.

[Kai02] M. Kaiser. Effizientes Beweisen mit einem formalen Beweissystem. Diplomarbeit (German), Fachbereich Mathematik, Universität Kaiserslautern, Germany, 2002.

[KL80] S. Kamin and J.-J. Levi. Two generalizations of the recursive path ordering. Technical report, Dep. of Computer Science, University of Illinois, Urbana, IL, 1980. Unpublished note.

[KMM00] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[KN94] D. Kapur and X. Nie. Reasoning about numbers in Tecton. In Z. W. Ras and M. Zemankova, editors, *Proc. of 8th International Symposium on Methodologies for Intelligent Systems (ISMIS'94)*, volume 869 of *LNCS*, pages 57–70. Springer, 1994.

[Knu81] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 2nd edition, 1981. Pages 326–328.

[KR90] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *AAAI*, pages 240–245, 1990.

[Kre65] G. Kreisel. Mathematical logic. In T. L. Saaty, editor, *Lectures on Modern Mathematics*, volume III, pages 95–195. Wiley, New York, 1965.

[KS96a] D. Kapur and M. Subramaniam. Automating induction over mutually recursive functions. In M. Wirsing and M. Nivat, editors, *AMAST*, volume 1101 of *LNCS*, pages 117–131. Springer, 1996.

[KS96b] D. Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *J. Autom. Reasoning*, 16(1–2):39–78, 1996.

[KS03] D. Kapur and M. Subramaniam. Automatic generation of simple lemmas from recursive definitions using decision procedures - preliminary report. In V. A. Saraswat, editor, *ASIAN*, volume 2896 of *LNCS*, pages 125–145. Springer, 2003.

[Küh00] U. Kühler. *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations*. PhD thesis, Universität Kaiserslautern, 2000.

[KW94] T. Kolbe and C. Walther. Reusing proofs. In *ECAI*, pages 80–84, 1994.

[KW97] U. Kühler and C.-P. Wirth. Conditional equational specifications of data types with partial operations for inductive theorem proving. In H. Comon, editor, *RTA*, volume 1232 of *LNCS*, pages 38–52. Springer, 1997.

[KZ95] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. of Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[LH02] B. Löchner and T. Hillenbrand. A phytography of WALDMEISTER. *AI Commun.*, 15(2–3):127–133, 2002.

[Löc04] B. Löchner. Things to know when implementing LPO. In S. Schulz, G. Sutcliffe, and T. Tammet, editors, *Proceedings of the 1st Workshop on Empirically Successful First Order Reasoning (ESFOR '04)*, ENTCS. Elsevier, 2004. Extended version to appear in *International Journal on Artificial Intelligence Tools*.

[LS01] R. Letz and G. Stenz. Model elimination and connection tableau procedures. In Robinson and Voronkov [RV01], pages 2015–2114.

[MM70] Z. Manna and J. McCarthy. Properties of programs and partial function logic. In *Machine intelligence 5*, pages 27–37. Edinburgh University Press, 1970.

[MSS96] J. P. Marques-Silva and K. A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *ICTAI*, pages 467–469, 1996.

[Mus80] D. R. Musser. On proving inductive properties of abstract data types. In *POPL*, pages 154–162, 1980.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[PP93] A. Pettorossi and M. Proietti. Rules and strategies for program transformation. In B. Möller, H. Partsch, and S. A. Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 263–304. Springer, 1993.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Pro94] M. Protzen. Lazy generation of induction hypotheses. In A. Bundy, editor, *CADE*, volume 814 of *LNCS*, pages 42–56. Springer, 1994.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[Ron04] R. Rondot. Integration von Entscheidungsprozeduren in den induktiven Theorembeweiser QuodLibet. Diplomarbeit (German), Fachbereich Informatik, Technische Universität Kaiserslautern, Germany, 2004.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov. Term indexing. In Robinson and Voronkov [RV01], pages 1853–1964.

[RV01] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[SBB+02] J. H. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.-P. Wirth, and J. Zimmer. Proof development with omega. In Voronkov [Vor02], pages 144–149.

[SBF+03] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, and M. Pollet. Proof development in OMEGA: The irrationality of square root of 2. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, volume 28 of *Kluwer Applied Logic series*, pages 271–314. Kluwer Academic Publishers, 2003. ISBN 1-4020-1656-5.

[Sho77] R. E. Shostak. On the sup-inf method for proving presburger formulas. *J. ACM*, 24(4):529–543, 1977.

[Sho84] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[SK97] J. Schumacher and U. Kühler. XQL — a graphical user interface for the inductive theorem prover QuodLibet. Unpublished technical report, Fachbereich Informatik, Universität Kaiserslautern, Germany, 1997.

[Spr96] C. Sprenger. Über die Beweissteuerung des induktiven Theorembeweisers QuodLibet mit Taktiken. Diplomarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, 1996.

[SS97] T. Schmidt-Samoa. Realisierung einer Taktik-basierten Beweissteuerungskomponente für den induktiven Theorembeweiser QuodLibet. Projektarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, Germany, 1997.

[SS98] G. Sutcliffe and C. B. Suttner. The tptp problem library - cnf release v1.2.1. *J. Autom. Reasoning*, 21(2):177–203, 1998.

[SS04] T. Schmidt-Samoa. *The New Standard Tactics of the Inductive Theorem Prover* QuodLibet. SEKI-Report SR–2004–01 (ISSN 1437–4447). SEKI, Saarland Univ., 2004. `http://www.ags.uni-sb.de/~cp/p/sr200401/welcome.html`.

[SS06a] T. Schmidt-Samoa. An even closer integration of linear arithmetic into inductive theorem proving. In J. Carette and W. M. Farmer, editors, *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005)*, volume 151, pages 3–20. ENTCS, 2006.

[SS06b] T. Schmidt-Samoa. Flexible heuristics for simplification with conditional lemmas by marking formulas as forbidden, mandatory, obligatory, and generous. *Journal of Applied and Non-Classical Logic: Implementation of logics*, 16(1–2):209–239, 2006. To appear.

[Ste99] G. L. Steele Jr. *Common Lisp—The Language*. Digital Press, 2nd edition, 1999.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.

[Str02] O. Strichman. On solving presburger and linear arithmetic with sat. In M. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *LNCS*, pages 160–170. Springer, 2002.

[SV93] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. In *ICTAI*, pages 48–55, 1993.

[TG05] R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Appl. Algebra Eng. Commun. Comput.*, 16(4):229–270, 2005.

[Vor02] A. Voronkov, editor. *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *LNCS*. Springer, 2002.

[Wal94] C. Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, J. A. Robinson, and J. H. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (2)*, pages 127–228. Oxford University Press, 1994.

[Wie03] F. Wiedijk. Comparing mathematical provers. In A. Asperti, B. Buchberger, and J. H. Davenport, editors, *MKM*, volume 2594 of *LNCS*, pages 188–202. Springer, 2003.

[Wir90] M. Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 675–788. Elsevier and MIT Press, 1990.

[Wir97] C.-P. Wirth. *Positive/Negative-Conditional Equations: A Constructor-Based Framework for Specification and Inductive Theorem Proving*, volume 31 of *Schriftenreihe Forschungsergebnisse zur Informatik*. Verlag Dr. Kovač, Hamburg, Arnoldstr. 49, D–27763 Hamburg, 1997. ISBN 3-86064-551-X, `http://www.ags.uni-sb.de/~cp/p/diss/welcome.html`.

[Wir04] C.-P. Wirth. Descente infinie + Deduction. *Logic Journal of the IGPL*, 12(1):1–96, 2004. `http://www.ags.uni-sb.de/~cp/p/d/welcome.html`.

[Wir05a] C.-P. Wirth. *Consistency of Recursive Definitions via Shallow Confluence of Non-Terminating Non-Orthogonal Conditional Term Rewriting Systems with any kind of Extra Variables*. SEKI-Report SR–2005–02 (ISSN 1437–4447). SEKI Publications, Saarland Univ., 2005. `http://www.ags.uni-sb.de/~cp/p/shallow/welcome.html`.

[Wir05b] C.-P. Wirth. History and future of implicit and inductionless induction: Beware the old jade and the zombie! In *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, number 2605 in LNAI, pages 192–203. Springer, 2005. `http://www.ags.uni-sb.de/~cp/p/zombie/welcome.html`.

[Zha95] H. Zhang. Contextual rewriting in automated reasoning. *Fundam. Inform.*, 24(1/2):107–123, 1995.

[ZKK88] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. L. Lusk and R. A. Overbeek, editors, *CADE*, volume 310 of *LNCS*, pages 162–181. Springer, 1988.

[ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In Voronkov [Vor02], pages 295–313.

Index

Curriculum Vitae

Personalien

Name:	Tobias Schmidt-Samoa
Anschrift:	Konrad-Adenauer-Str. 5 67663 Kaiserslautern
Geburtsdatum:	30.10.1973
Geburtsort:	Eschwege
Familienstand:	verheiratet, keine Kinder
Staatsangehörigkeit:	deutsch

Ausbildung

1979-1983	Grundschule Alexander-von-Humboldt-Schule, Eschwege
1983-1989	Gymnasium Leuchtbergschule, Eschwege
1989-1992	Oberstufengymnasium, Eschwege Abschluss: Abitur
1992-2000	Studium der Informatik mit Nebenfach Mathematik, Universität Kaiserslautern Abschluss: Diplom-Informatiker

Berufstätigkeit

1.4.2000-31.3.2003	Angestellter beim Sonderforschungsbereich 501 “Entwicklung großer Systeme mit generischen Methoden”, Universität Kaiserslautern
1.4.2003-31.3.2005	Wissenschaftlicher Mitarbeiter, Fachbereich Informatik, Technische Universität Kaiserslautern