A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations

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

Ulrich Kühler

Berichterstatter:	Prof. Dr. Jürgen Avenhaus
	Prof. Dr. Klaus E. Madlener
Dekan:	Prof. Dr. Michael M. Richter

Datum der wissenschaftlichen Aussprache: 22. Oktober 1999

D 386

Abstract

Data types such as the natural numbers, lists, strings, trees, graphs etc. are essential for the design and implementation of most software systems. A given data type D can often be adequately formalized using an equational specification *spec* with inductive semantics (i.e. the carriers of the models associated with *spec* as the semantics of *spec* are inductively defined). This entails that statements expressing valid properties of the operations of D are represented by the inductive theorems of the specification *spec*. Therefore, *inductive theorem proving* constitutes the basis of a suitable formal method for reasoning about data types.

The subject of this thesis is an inductive theorem prover (named QuodLibet) which we have developed to attain two essential goals. Firstly, our prover is applicable to data types with *partial* operations, although data types like these lead to specifications that need not be sufficiently complete or may induce non-terminating rewrite relations. Secondly, by offering a high degree of flexibility with regard to the supported forms of proof control, the *tactic*-based approach to the problem of proof control developed for QuodLibet yields a *practically* useful compromise between interactive proof control on the one hand and automated proof control on the other hand.

In the first part of this thesis, we present a comprehensive and rewrite-based *formal framework for inductive theorem proving* which serves as the logical foundations of QuodLibet. The main components of this framework are (i) an algebraic specification language for the formalization of data types, (ii) a "semantic" induction order for guaranteeing applicability of induction hypotheses, (iii) a calculus for inductive proofs for formal reasoning about data types and (iv) a concept of a so-called proof state graph for the adequate representation of proof constructions. In particular, the specification language allows axioms to be conditional equations (or rewrite rules) with positive and negative conditions, and it has precisely defined *admissibility conditions*, which can be easily verified for most practically relevant specifications (as no proofs of termination are required). Moreover, by incorporating proof state graphs, our framework is capable of supporting the delayed or *lazy* computation of induction hypotheses, *mutual* induction for conjectures about mutually recursive operations as well as *multiple* complete or incomplete proof attempts for the same conjecture.

The second part of this thesis deals with QuodLibet as a software system. We sketch the software architecture and describe the functionality of the system (which can be utilized through a command interpreter or a graphical user interface). The focus of the second part, however, is on the approach to the problem of proof control developed for QuodLibet. This approach is based on so-called (proof) *tactics*, i.e. proof control routines written in a special proof control language named QML. QuodLibet provides, as part of the approach, a set of tactics (in addition to the elementary inference rules), which range from tactics for trivial simplification steps to tactics representing comprehensive inductive *proof procedures*. Moreover, the system allows new tactics (written by the user in QML) to be integrated into the system to enhance its performance. Finally, we supply experimental evidence for our claim that the desired (partial) automation of the proof control of QuodLibet has been achieved to a considerable extent.

Zusammenfassung

Datentypen wie etwa die natürlichen Zahlen, Listen, endliche Graphen usw. sind von großer Bedeutung für den Entwurf und die Implementierung von Softwaresystemen. Ein Datentyp D läßt sich oft in Form einer Gleichungsspezifikation *spec* mit *induktiver* Semantik (d.h. die Träger der Modelle, die die Semantik von *spec* ergeben, sind induktiv definiert) angemessen formalisieren, weswegen Aussagen über gültige Eigenschaften der Operationen von D von den induktiven Theoremen der Spezifikation *spec* repräsentiert werden. Infolgedessen bildet das Gebiet des *induktiven Theorembeweisens* die Grundlage einer geeigneten *formalen* Methode für die Behandlung von Datentypen.

Das Thema der vorliegenden Dissertation ist ein induktiver Theorembeweiser (genannt QuodLibet), bei dessen Entwicklung wir zwei wesentliche Ziele erreicht haben. Zum einen ist der Beweiser anwendbar auf Datentypen mit *partiellen* Operationen, die in der Regel zu unvollständigen oder "nicht-terminierenden" Spezifikationen führen. Zum anderen bietet QuodLibet einen hohen Grad an Flexibilität bzgl. der unterstützten Formen von Beweissteuerung. Sein *Taktik*-basiertes Beweissteuerungskonzept stellt einen für die Praxis geeigneten Kompromiss zwischen interaktiver und automatisierter Beweissteuerung dar.

Im ersten Teil dieser Dissertation präsentieren wir einen umfassenden rewrite-basierten *formalen Rahmen für induktives Theorembeweisen*, der die theoretischen Grundlagen von QuodLibet bildet. Die wesentlichen Bestandteile dieses Rahmens sind (i) eine algebraische Spezifikationssprache für die Formalisierung von Datentypen, (ii) eine Induktionsordnung, mit der die Anwendbarkeit von Induktionshypothesen sichergestellt werden kann, (iii) ein Kalkül für Induktionsbeweise für formale Beweisführungen und (iv) ein Konzept eines Beweiszustandsgraphen für die angemessene Repräsentation von Beweiskonstruktionen. Insbesondere erlaubt die Spezifikationssprache positiv/negativ bedingte Gleichungen (oder Rewrite-Regeln) als Axiome und sieht präzise definierte *Zulässigkeitsbedingungen* vor, die für viele relevante Spezifikationen effektiv verifiziert werden können (da keine Terminationsbeweise erforderlich sind). Weiter unterstützt der Rahmen auf Grund der Verwendung von Beweiszustandsgraphen die verzögerte Berechnung von Induktionshypothesen, wechselseitige Induktion und parallele Beweisversuche für dieselbe Behauptung.

Der zweite Teil dieser Dissertation hat das Softwaresystem QuodLibet zum Thema. Wir skizzieren zunächst die Softwarearchitektur und beschreiben die Funktionalität des Systems (die über einen Kommando-Interpreter oder eine graphische Benuztungsoberfläche genutzt werden kann). Den Schwerpunkt des zweiten Teils bildet jedoch der für QuodLibet entwickelte Beweissteuerungsansatz. Dieser Ansatz beruht auf sogenannten (Beweis-) *Taktiken*, d.h. Beweissteurungsroutinen, die in der eigens entwickelten Beweissteuerungssprache QML geschrieben sind. QuodLibet stellt einen größeren Vorrat an Taktiken zur Verfügung (zusätzlich zu den elementaren Inferenzregeln), u.a. solche zur Realisierung vollständiger *induktiver Beweisverfahren*, mit der die geforderte (partielle) Automatisierung der Beweissteuerung von QuodLibet in vielen Fällen erreicht wird. Daneben kann der Benutzer die Leistungsfähigkeit des Beweisers durch die Bereitstellung neuer, in QML geschriebener Taktiken steigern.

Contents

1 Introduction and Overview **1**

I A Formal Framework for Inductive Theorem Proving **3**

2 Preliminaries **5**

2.1 Overview of the Formal Framework 5

2.2 Requirements for the Formal Framework 7

2.2.1 The Specification Language 7

2.2.2 The Induction Order 9

2.2.3 Inference Rules 10

2.2.4 Representation of Proof Constructions 13

2.3 An Example of a Proof Construction 14

2.4 Basic Notions and Notations 16

3 The Specification Language **19**

3.1 Specifications with Constructors 19

3.1.1 Syntax of Specifications with Constructors 20

3.1.2 Model Semantics of Specifications with Constructors 21

3.1.3 Data Models 23

3.2 Admissibility Conditions 24

3.2.1 Positive/Negative Conditional Rewrite Specifications 24

3.2.2 The Standard Data Model $\mathcal{M}(\mathit{spec})$ 27

3.3 A Confluence Criterion 28

3.4 Inductive Validity 31

4 The Induction Order **33**

4.1 $\mathcal{A}$-Counterexamples and Inference Rules 33
4.2 Weights and the Semantic Induction Order $\precsim_{\mathcal{A}}$ 36
4.3 Clauses with Order Literals 39

5 Inference Rules 43
5.1 Preliminaries 44
5.2 Non-Applicative Inference Rules 45
5.2.1 Establishing Simple Tautologies 45
5.2.2 Decomposing Atoms 46
5.2.3 Removing Redundant Literals 50
5.2.4 Making Use of Negative Literals 52
5.2.5 Further Inference Rules for Order Atoms 53
5.2.6 Non-Applicative Case Analyses 56
5.3 Applicative Inference Rules 57
5.3.1 Non-Inductive Applicative Inference Rules 58
5.3.2 Inductive Applicative Inference Rules 63
5.4 Concluding Remarks 64

6 Proof State Graphs 67
6.1 Representing Proof Constructions 67
6.2 Soundness Results 76
6.2.1 Soundness 76
6.2.2 Refutational Soundness 78
6.3 Concluding Remarks 80

II The Inductive Theorem Prover QuodLibet 83

7 An Introduction to QuodLibet 85
7.1 Requirements 86
7.2 The System Structure 89
7.3 The Command Language 91
7.3.1 Commands for Formalizing Data Types 92
7.3.2 Commands for Constructing Proof State Trees 96
7.3.3 Miscellaneous Commands 102

7.3.4 An Example of an Interactive Proof Construction 103
7.4 The Graphical User Interface 113
7.5 Concluding Remarks 119

8 Tactic-Based Proof Control in QuodLibet 121
8.1 The Overall Idea 122
8.2 The Proof Control Language QML 126
8.2.1 Essentials of QML 126
8.2.2 Commands Related to QML 136
8.3 QML Routines Provided by QuodLibet 140
8.3.1 The Proof Control Data Base 141
8.3.2 Simplifying Clauses in Goals 154
8.3.3 Constructing Inductive Case Analyses 160
8.3.4 Inductive Proof Strategies 166
8.4 Concluding Remarks 172

9 Conclusion 173

A The Proofs 175

B Syntax of the Command Language of QuodLibet 197

C The Inference Rules of QuodLibet 205
C.1 Non-Applicative Inference Rules 206
C.1.1 Establishing Simple Tautologies 206
C.1.2 Decomposing Atoms 206
C.1.3 Removing Redundant Literals 207
C.1.4 Making Use of Negative Literals 208
C.1.5 Further Inference Rules for Order Atoms 209
C.1.6 Non-Applicative Case Analyses 210
C.2 Applicative Inference Rules 211
C.2.1 Non-Inductive Applicative Inference Rules 211
C.2.2 Inductive Applicative Inference Rules 212

D Syntax of QML 215

E Examples of Proof Constructions **221**

E.1 Truth Values and Natural Numbers 221

E.1.1 The Basic Specification 221

E.1.2 Addition and Multiplication 225

E.1.3 (Partial) Subtraction and Division 229

E.1.4 Ackermann's Function 231

E.1.5 A Non-Terminating Operation 232

E.2 Lists of Natural Numbers 234

E.2.1 Basic Definitions 234

E.2.2 Some Operations on Lists 234

E.2.3 Mergesort 238

E.3 Binary Trees with Natural Numbers 244

E.3.1 Basic Definitions 244

E.3.2 Search Trees 244

E.3.3 A Flatten Operation 247

Bibliography **251**

Chapter 1

Introduction and Overview

Data types such as the natural numbers, lists, strings, trees, graphs etc. are essential for the design and implementation of most software systems. A collection D of data domains and operations on these data domains is usually called a *data type* if all data items in the data domains of D are finitely generated by the operations of D. It is commonly accepted that a given data type D can often be adequately formalized using an equational specification *spec* with inductive semantics (i.e. the carriers of the models associated with *spec* as the semantics of *spec* are inductively defined). This entails that statements expressing valid properties of the operations of D are represented by the inductive theorems of the specification *spec*. Therefore, *inductive theorem proving* constitutes the basis of a suitable formal method for reasoning about data types.

The subject of this thesis is an inductive theorem prover (named QuodLibet) which we have developed to attain two essential goals. Firstly, our prover is intended to be applicable to data types with *partial* operations. The major problem caused by this requirement is that "natural" formalizations of data types with partial operations often lead to specifications that need not be sufficiently complete or may induce non-terminating rewrite relations. Secondly, we have sought to come up with a novel approach to the problem of proof control for our inductive theorem prover that is to be characterized by a high degree of flexibility with regard to the forms of proof control it supports. The underlying expectation related to this requirement is that an approach to proof control like this should yield a *practically* useful compromise between interactive proof control on the one hand and automated proof control on the other hand.

The thesis is organized as follows. In the first part, which comprises Chapters 2 – 6, we present a comprehensive and rewrite-based *formal framework for inductive theorem proving* which serves as the logical foundations of QuodLibet. The main components of this framework are (i) an algebraic specification language for the formalization of data types, (ii) a "semantic" induction order for guaranteeing applicability of induction hypotheses, (iii) a calculus for inductive proofs for formal reasoning about data types and (iv) a concept of a so-called proof state graph for the adequate representation of proof constructions.

In Chapter 2, we first summarize the essential features of the proposed formal framework in a brief overview. Then we motivate the ensuing chapters of Part I by discussing some of the general requirements which have guided the development of the framework. After that, a simple example of a proof construction is given that should convey the overall idea of how one can prove inductive theorems within our framework.

The first part of Chapter 3 deals with the syntax and model semantics of our so-called specifications with constructors. Besides, the notion of a data model is introduced which serves as a basis for a suitable (total) algebra semantics. In 3.2 we define the admissibility conditions of our specification language. For this purpose, we associate a rewrite relation with every specification with constructors. Since admissibility of a specification requires confluence of the associated rewrite relation, we provide an easily verifiable confluence criterion in 3.3 that does not presuppose termination of the rewrite relation. In 3.4 finally, we define an appropriate notion of inductive validity with respect to an admissible specification.

In Chapter 4, we first introduce the notion of an $\mathcal{A}$-*counterexample* (for a clause in a data model $\mathcal{A}$ of the specification *spec*). Furthermore, we give an abstract characterization of the induction order $\precsim_{\mathcal{A}}$ associated with every data model $\mathcal{A}$, specify the general form of inference rules and define the soundness of inference rules. Thereafter (in 4.2), a concrete *semantic* induction order is proposed. In 4.3 finally, we extend clauses as defined in the preceding chapter by so-called *order literals*.

In Chapter 5, we present the (twenty-five) inference rules of our calculus for inductive proofs, which constitute the setting for the *syntactic* reasoning used in proofs of inductively valid clauses within the proposed formal framework for inductive theorem proving.

In Chapter 6, we first give a formal definition and examples of proof state graphs and related notions. Thereafter (in 6.2), we present the major soundness results of this thesis.

The overall subject of the second part of this thesis, which consists of Chapters 7 and 8, is the *software system* QUODLIBET.

In 7.1 we begin our introduction to QUODLIBET by describing general requirements for an inductive theorem prover that is to be based on the formal framework discussed in the first part of this thesis. After that (in 7.2), we briefly explain the high level system structure of the software system QUODLIBET. Section 7.3 then deals with the functionality of the prover in terms of its command language, while essential properties of QUODLIBET's graphical user interface are presented in 7.4.

In Chapter 8, we first give a motivating overview of our approach to proof control (in 8.1). Then we deal with the essentials of the proof control language QML in 8.2. Moreover, a description of the system commands related to QML is given. Thereafter (in 8.3), we present the tactics — or rather (QML) proof control routines — which we provide as part of QUODLIBET and which realize the required (partial) automation of the proof control of our inductive theorem prover.

In Chapter 9 finally, we summarize the work presented in this thesis.

Part I

A Formal Framework for Inductive Theorem Proving

Chapter 2

Preliminaries

The subject of this first major part of the thesis is a comprehensive rewrite-based and user-oriented formal framework for inductive theorem proving, which was designed to serve as the logical or theoretical foundations of QuodLibet. The main components of the framework are

- an algebraic *specification language* for the formalization of data types
- an *induction order* for guaranteeing applicability of induction hypotheses
- a *calculus for inductive proofs* for formal reasoning about data types and
- a *concept of a* so-called *proof state graph* for the adequate representation of proof constructions.

In this introductory chapter to Part I, we first summarize the essential features of the framework in a brief overview (2.1). Then we motivate the ensuing chapters of Part I by discussing some of the general requirements which have guided the development of the framework (2.2). After that, a simple example of a proof construction is given that should convey the overall idea of how one can prove inductive theorems within our framework (2.3). The chapter concludes with a compilation of the basic notions and notations used throughout the remaining chapters of the thesis (2.4).

2.1 Overview of the Formal Framework[1]

An algebraic (i.e. equational) *specification language* for the formalization of data types with partial operations constitutes the basis of our framework for inductive theorem proving. The specification language can be described by its syntax, its (inductive) semantics and its admissibility conditions. While the syntax determines the signature and the set of axioms admitted in a specification, the semantics indicates what particular model class is to be associated with a specification as its meaning, and the

[1]This section may be skipped on a first reading.

admissibility conditions have to guarantee that the semantics is actually meaningful, i.e. the associated model class is not empty.

Our specification language requires constructor symbols for each sort in a signature so that all data items in a data type can be designated with constructor ground terms, and provides *constructor variables* which range over data items only. Furthermore, *conditional* equations (or rewrite rules) with positive and negative conditions are admitted as axioms in specifications.

In addition to the usual notion of a model we define so-called *data models* that serve as a basis for a suitable (total algebra) semantics of specifications. For every *admissible* specification *spec* the unique existence of a "best" data model $\mathcal{M}(spec)$ is guaranteed — "best" in the sense that $\mathcal{M}(spec)$ has interesting algebraic properties resembling those of the initial model of a usual equational specification. The appropriateness of our semantics is underlined by an important monotonicity result: Contrary to initial algebra semantics, the extension of an admissible specification in a "consistent" way does not result in the loss of inductive theorems. In other words, every formula valid in the class of all data models of the original specification remains valid in the class of all data models of the extended specification.

Essentially, admissibility of a specification with constructors *spec* means that the rewrite relation which we associate with the positive/negative conditional rewrite rules in *spec* is confluent. Since we are also interested in formalizing data types with non-terminating operations, we provide a confluence criterion which does not presuppose termination of the rewrite relation but is based on simple syntactic properties of the specifying rewrite system. As a consequence, we obtain easily testable admissibility conditions that are fulfilled by many practically relevant specifications, including such that contain *incomplete* or even *non-terminating* axiomatizations of partial operations of data types.

The formal counterpart of a "true" statement about the operations of a data type (e.g. a verification condition) is an inductive theorem. A formula is called an *inductive theorem* or *inductively valid* with respect to an admissible specification *spec* if it is valid in the class of models associated with *spec* as its semantics, which is the class of data models of *spec*.

The calculus for inductive proofs belonging to the proposed framework is to determine the basic setting for the *syntactic* reasoning used in the construction of formal proofs for inductive theorems. It is based on the notion of inductive validity defined by the specification language and on a well-founded induction order. Speaking in simplified terms, the *induction order* is needed for guaranteeing that induction hypotheses are actually "smaller" (in an appropriate sense) than the formulas they are applied to. To obtain a more flexible and powerful induction order, we require so-called *weights* to be associated with formulas. A weight w in a *goal* $\langle \Gamma\,;\, w \rangle$ is to provide a measure of the "size" of the formula Γ for comparisons w.r.t. the induction order.

The presented *calculus for inductive proofs* restricts inductive theorems to first-order equational clauses. Each inference rule of our calculus reduces a goal $\langle \Gamma\,;\, w \rangle$ to a set of new (sub-) goals. For the reduction of $\langle \Gamma\,;\, w \rangle$, an inference rule may make use of further goals to be applied to $\langle \Gamma\,;\, w \rangle$ either as *induction hypotheses* or as (possibly unproved) *lemmas*. All in all, the comprehensive and expressive system of inference rules is adapted to the kind of reasoning users do in informal arguments, but also to the needs of proof automation.

In case a goal is applied as induction hypothesis, a certain *order condition* must be fulfilled, i.e. the induction hypothesis must be smaller (w.r.t. the induction order) than the goal to which the induction hypothesis is applied. Since verifications of such order conditions often involve inductive proofs, we *integrate the induction order into our calculus* by (1) extending clauses with so-called *order literals* and by (2) providing specific inference rules for proving clauses with order literals. Consequently, we are able to represent and verify these order conditions in our calculus for inductive proofs.

As a further major feature of our framework for inductive theorem proving, a novel concept of a so-called *proof state graph* is proposed. A proof state graph resembles an AND/OR graph, and its purpose is to represent the relations on the set of (sub-) goals in a proof construction that determine the states of the various proof attempts. Essentially, these relations arise from reductions of goals to subgoals and from applications of goals to other goals as induction hypotheses or lemmas. Besides facilitating a useful criterion for inductive validity applicable to *any* goal in a possibly incomplete proof attempt, proof state graphs also support the delayed or *lazy* computation of induction hypotheses, mutual induction, applications of *unproved* lemmas, and *multiple* complete or incomplete proof attempts for the same conjecture.

2.2 Requirements for the Formal Framework

We are now going to provide the general motivation for the remaining chapters of Part I in a *coherent* form in that we compile the major requirements for the *entire* formal framework in this section. In accordance with the above mentioned composition of the framework, we discuss requirements with regard to the specification language (2.2.1), the induction order (2.2.2), the system of inference rules (2.2.3) and the concept of a proof state graph for the representation of proof constructions (2.2.4).

2.2.1 The Specification Language

The subsequent two elementary examples are intended to motivate the major requirements which have lead to the specification language presented in Chapter 3.

Example 2.2.1 Let D be a data type that comprises the natural numbers as its data domain and the division along with other arithmetic operations (see below). The following set E of conditional equations could be the set of axioms in an algebraic specification $spec_{\mathsf{div}} = (sig, C, E)$ of D (C denotes the *constructors* in sig).

$$\begin{array}{ll}\mathsf{plus}(x,\mathsf{0}) & = x\\ \mathsf{plus}(x,\mathsf{s}(y)) & = \mathsf{s}(\mathsf{plus}(x,y))\end{array}$$

$$\begin{array}{ll}\mathsf{times}(x,\mathsf{0}) & = \mathsf{0}\\ \mathsf{times}(x,\mathsf{s}(y)) & = \mathsf{plus}(\mathsf{times}(x,y),x)\end{array}$$

$$\begin{array}{ll}\mathsf{minus}(x,\mathsf{0}) & = x\\ \mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) & = \mathsf{minus}(x,y)\end{array}$$

$$\begin{array}{ll}\mathsf{less}(x,\mathsf{0}) & = \mathsf{false}\\ \mathsf{less}(\mathsf{0},\mathsf{s}(y)) & = \mathsf{true}\\ \mathsf{less}(\mathsf{s}(x),\mathsf{s}(y)) & = \mathsf{less}(x,y)\end{array}$$

$$\begin{array}{lll}\mathsf{div}(x,y) = \mathsf{0} & \leftarrow & y \neq \mathsf{0} \ \wedge\ \mathsf{less}(x,y) = \mathsf{true}\\ \mathsf{div}(x,y) = \mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),y)) & \leftarrow & y \neq \mathsf{0} \ \wedge\ \mathsf{less}(x,y) = \mathsf{false}\end{array}$$

Although E yields an appropriate axiomatization of the data type D, the axioms in E (or their respective formulations) are not admissible w.r.t. the specification formalisms of various first-order frameworks for inductive theorem proving:

Firstly, $spec_{\mathsf{div}} = (sig, C, E)$ is not *sufficiently complete* w.r.t. the constructors true, false, 0 and s in C (see e.g. Wirsing, 1990), since neither minus nor div are completely defined by E. In the specification formalisms of inductive theorem provers such as SPIKE (Bouhoula & Rusinowitch, 1995), NQTHM (Boyer & Moore, 1979) or INKA (Hutter & Sengler, 1996), however, each non-constructor operation must be completely defined, i.e. for a *partial* operation (such as the subtraction or division) some of its total extensions has to be axiomatized.

Secondly, the rewrite-based specification languages of SPIKE and RRL (Kapur & Subramaniam, 1996c) require the left-hand side of each conditional equation to be greater than any other term in this conditional equation w.r.t. a reduction order. Since $\mathsf{div}(x,y)$ is *smaller* than $\mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),y))$ w.r.t. any simplification order (see Dershowitz, 1987), it is fairly difficult to prove such admissibility of E.

Thirdly, two conditional equations in E each contain a *negative* condition, namely $y \neq \mathsf{0}$. This is ruled out by Bouhoula and Rusinowitch (1995) for instance.

Example 2.2.2 *(Example 2.2.1 continued)* A tail-recursive variant of div that is non-terminating but "efficient" on its domain can be axiomatized as follows:

$$\begin{array}{lll}\mathsf{div1}(x,y,z_1,z_2) = z_1 & \leftarrow & x = z_2\\ \mathsf{div1}(x,y,z_1,z_2) = \mathsf{div1}(x,y,\mathsf{s}(z_1),\mathsf{plus}(z_2,y)) & \leftarrow & x \neq z_2\end{array}$$

If $n > 0$ then $\mathsf{div1}(\mathsf{s}^{n \cdot m}(\mathsf{0}),\mathsf{s}^n(\mathsf{0}),\mathsf{0},\mathsf{0})$ can be evaluated to $\mathsf{s}^m(\mathsf{0})$ with E', where E' consists of E and the two axioms for div1. Hence, the equational clause

$$y = \mathsf{0} \ \vee\ \mathsf{div1}(\mathsf{times}(y,z),y,\mathsf{0},\mathsf{0}) = z \tag{2.1}$$

formalizes an intuitively "true" statement and should be valid in the class of models (i.e. the semantics) associated with the specification $spec_{\mathsf{div1}} = (sig', C, E')$.

Due to their requirement that axiomatizations of operations be "terminating", none of the frameworks for inductive theorem proving described by Bouhoula and Rusinowitch (1995), Boyer and Moore (1979), Hutter and Sengler (1996) or Kapur and Subramaniam (1996c) accept E' (or its respective formulations) as an admissible set of axioms.

All in all, we seek to provide a specification language that

- facilitates adequate formalizations of data types with *partial* operations including those calling for incomplete or even non-terminating specifications
- admits as axioms in specifications conditional equations (or rewrite rules) with positive as well as *negative* conditions and
- has precisely defined admissibility conditions which can be easily verified for most practically relevant specifications.

2.2.2 The Induction Order

As mentioned above, we are going to define a well-founded induction order as part of the foundations of our calculus for inductive proofs (see Chapter 4). For a discussion of the requirements related to the induction order let us again consider the specification $spec_{\mathsf{div1}}$ from Example 2.2.2, which also demonstrates some of the problems resulting from the expressive power of a specification language as required above.

Since we intend to admit specifications defining "non-terminating" operations (such as div1), it is only natural that we are interested in an induction order allowing proofs of inductive theorems which express *properties of non-terminating operations* (such as clause (2.1)). Now due to our requirements for the specification language, we cannot expect the rewrite relation $\longrightarrow_R$ associated with an admissible specification to be terminating. Hence our induction order cannot be simply based on a (syntactic) *reduction order* extending $\longrightarrow_R$ — as is the case in the rewrite-based frameworks of inductive theorem provers such as RRL (Kapur & Subramaniam, 1996c) and Spike (Bouhoula & Rusinowitch, 1995) and in the rewrite-based frameworks described by Reddy (1990) or by Bronsard, Reddy, and Hasker (1994).

A further, practically important requirement is that the induction order be suited for proofs by *destructor induction.* As div is axiomatized in Example 2.2.1 by what is commonly called *destructor recursion*,[2] an inductive theorem of the form $\Gamma(\,\mathsf{div}(x,y)\,)$ typically has to be proved by destructor induction. For a proof of this kind, one needs an induction order with respect to which, informally speaking, the induction hypothesis $\Gamma(\,\mathsf{div}(\mathsf{minus}(x,y),y)\,)$ is "smaller" than the formula $\Gamma'(\,\mathsf{div}(x,y)\,)$ it is applied to. Here, $\Gamma'(\,\mathsf{div}(x,y)\,)$ stands for a formula derived from $\Gamma(\,\mathsf{div}(x,y)\,)$ and *governed* by the literals $\mathsf{less}(x,y) \neq \mathsf{true}$ and $y \neq \mathsf{0}$ (see e.g. Example 4.3.1). Note that in general, proofs by destructor induction are fairly difficult (if not practically impossible) in the

[2] after Boyer and Moore (1979); see Section 4.3

rewrite-based frameworks of RRL and SPIKE. This is mainly due to the fact that induction orders based on (syntactic) reduction orders are used there.[3]

2.2.3 Inference Rules

Because of several case studies in the literature (see below) and our own experience of automated inductive theorem provers, such as NQTHM, INKA, RRL, SPIKE, UNICOM (Gramlich & Lindner, 1991) etc., we have come to adopt a rather pragmatic view of inductive theorem proving:

> Successfully employing an inductive theorem prover in "real-life" problem domains has not yet been possible without a knowledgeable human user who can interact with the system on various levels.

Accordingly, we demand that even the development of a calculus for inductive proofs should begin with a clear emphasis on *user interaction*, whereas automated proof control is regarded as a long-term goal. In the following we are going to give reasons for this before discussing more specific requirements for the design of our calculus for inductive proofs.

Anyone with practical experience of inductive theorem proving is certainly aware of the empirical fact that proofs of non-trivial inductive theorems normally require the use of additional *lemmas*. In the context of mathematical induction, a lemma for an inductive theorem can be thought of as any inductively valid formula that is useful for a proof of the theorem but not an axiom. Roughly speaking, lemmas contribute necessary knowledge of the problem domain (i.e. the given data type) to the proofs of inductive theorems. Consequently, most calculi for inductive proofs provide inference rules allowing the use of lemmas. Once determined lemmas can usually be applied like axioms to simplify formulas by rewriting or subsumption, or may, for instance, give rise to case analyses, support generalization steps etc. That the need of additional and separate lemmas in induction proofs is indeed in the nature of inductive theorem proving is underpinned by the fact that, in contrast to deductive theorem proving, applications of Gentzen's Cut rule cannot be eliminated in inductive proofs (see Bibel & Eder, 1993).

Undoubtedly, the determination of useful lemmas for inductive theorems still poses one of the most difficult problems in the field of inductive theorem proving. There are a few first approaches to automated lemma speculation in the literature, e.g. by Boyer and Moore (1979), Hutter (1990), Walsh (1994), Ireland and Bundy (1996) and Kapur and Subramaniam (1996b), but the following simple example may indicate what kinds of difficulty will have to be overcome before missing lemmas can be discovered mechanically. For some inductive theorems, even the formulation of suitable lemmas is not possible without an extension of the specification by new operations and axioms:

[3] In this context it is interesting to observe (again) that $\mathsf{div}(\mathsf{minus}(x,y),y)$ is *bigger* than $\mathsf{div}(x,y)$ w.r.t. any simplification order.

Example 2.2.3 Consider a specification of the natural numbers with constructors $\mathsf{0}$ and s and an operation double defined by the equational axioms

$$\begin{aligned} \mathsf{double}(\mathsf{0}) &= \mathsf{0} \\ \mathsf{double}(\mathsf{s}(x)) &= \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \end{aligned}$$

Obviously, the operation double has no positive fix-points, which means that the clause $x = \mathsf{0} \ \vee \ \mathsf{double}(x) \neq x$ is an inductive theorem of the specification. Still, we cannot imagine how to find a formal proof of this theorem unless additional function symbols besides $\mathsf{0}$, s, double may occur in the lemmas to be used.

However, by *extending the specification* with (i) a sort bool (with true and false as constructors), (ii) an operation less for the usual well-founded order on the natural numbers and (iii) suitable axioms for less (as in Example 2.2.1), we can obtain a proof. For then it is no longer difficult — at least for someone with an understanding of the problem domain — to discover and prove the missing lemmas, such as for instance

$$x = \mathsf{0} \ \vee \ \mathsf{less}(x, \mathsf{double}(x)) = \mathsf{true} \qquad \text{and} \qquad \mathsf{less}(x, y) \neq \mathsf{true} \ \vee \ x \neq y$$

and complete the proof of the theorem.

Moreover, some extensive case studies on inductive theorem proving (Boyer & Moore, 1979; Gramlich, 1990; Zhang & Hua, 1992) show that to prove a non-trivial inductive theorem with an *automated* inductive theorem prover, the user himself[4] has to discover most of the (key) lemmas needed for a proof. In order to be able to do that, he is normally compelled to develop a more or less detailed sketch of the proof *by hand*. Besides, these case studies further imply that user intervention may also be necessary for the generation of induction schemes in those cases where the heuristics based on recursion analysis fail, and for the control of rewriting in the simplification of formulas when the set of rewrite rules is not confluent. Furthermore, Boyer and Moore (1990) emphasize the importance of the user in finding proofs. They admit that their inductive theorem prover NQTHM, when applied to complex problems, should be seen as a *proof checker* rather than an automated theorem prover. We have a similar view of the capabilities of automated inductive theorem provers and the resulting need of user interaction. In contrast to some of the designers of earlier automated inductive theorem provers, however, we conclude that the design of the formal framework for an inductive theorem prover should not be solely determined by the objective to easily automate the construction of proofs. Instead, we demand that even on the (low) level of the calculus for inductive proofs, the *necessity* of the user interacting with the theorem prover must be taken into account as well.

All in all, the following more specific requirements have guided the design of (the inference rules of) our calculus for inductive proofs:

- **User-orientation:** As mentioned above, proving non-trivial inductive theorems with an inductive theorem prover usually involves a knowledgeable human user

[4] or *her*self

who may have to interact with the prover in various ways. For that purpose, it is often crucial that the user can achieve a good understanding of the state of the current (possibly failed) proof attempt. This can be facilitated by providing a calculus for inductive proofs which is oriented towards human proof techniques and easily comprehensible to the user — just like natural deduction or sequent calculi and unlike resolution and paramodulation calculi in the case of deductive theorem proving. Moreover, the calculus should be *goal-directed* (see Wirth & Kühler, 1995) meaning that every proof problem of a proof attempt is connected with the conjecture to be proved (in the graphical representation of the proof attempt). Besides, user acceptance of the calculus can be improved by a concept of an induction hypothesis that matches the user's idea of (textbook) proofs by mathematical induction. In particular, there should be inference rules for applications of formulas as *explicit* induction hypotheses. As a consequence, induction hypotheses need not be guessed long before they are (possibly) applied, as is suggested by Boyer and Moore (1979), Walther (1994) or Kapur and Subramaniam (1996c).[5]

- **Expressiveness:** For the successful construction of sophisticated proofs with an inductive theorem prover, it may be necessary for the user to be able to translate his hand-made proof sketches into concrete inference steps and make the prover execute these (see e.g. Boyer & Moore, 1988; Hua & Zhang, 1992). Therefore, the calculus for inductive proofs should also comprehend a variety of certain *expressive* inference rules that formalize important human proof techniques but lead to infinite branching points in the search space, thus making automated proof control more difficult. Examples of such inference rules are rules for case analyses or explicit instantiations of lemmas and induction hypotheses. Furthermore, we require the calculus to also comprise inference rules for fairly elementary proof steps so that the user can compel the prover (in difficult situations) to follow his proof ideas as closely as possible.

- **Automation:** Nevertheless, we think that to be acceptable to users, an inductive theorem prover needs to be capable of taking care of "routine work" (i.e. simple lemmas or proof obligations) without user intervention. To facilitate the (partial) automation of constructing proofs within the proposed framework, important restrictions on the specification language and the formulas used for representing inductive theorems appear to be reasonable: Neither (i) higher-order variables nor (ii) predicate symbols (except for '=')[6] nor (iii) explicitly quantified formulas should be admitted; and (iv) axioms as well as inductive theorems should be restricted to (equational) clause form. (Refer to Gordon and Melham (1993) e.g. for an opposite approach.)

[5] Protzen (1994) proposes another calculus for inductive proofs that allows the "lazy" (i.e. delayed) generation of induction hypotheses.

[6] In many-sorted specifications with inductive semantics, predicate symbols can be easily replaced with function symbols that have a sort bool for the two truth values as their result sorts.

- **Refutational soundness:** In goal-directed settings, an attempt at constructing a proof for a *false* conjecture may result in the deduction of an *obviously* contradictory formula, such as the empty clause. Now in order to make *safe* use of such situations, we require our calculus for inductive proofs to be *refutationally sound*. By this we mean that, informally stated, we can correctly infer the falseness of a conjecture Γ from the existence of a proof attempt for Γ that contains the empty clause. Since practical experience shows that (initial) formalizations of data types usually contain a lot of specification errors which result in false conjectures, we consider refutational soundness a practically useful property. Note that there are calculi for inductive proofs, for instance the ones underlying the inductive theorem provers NQTHM and INKA, that are not refutationally sound.

2.2.4 Representation of Proof Constructions

We have said before that we seek to provide a concept of a so-called proof state graph for representing the dependencies among the (sub-) goals in a proof construction which determine the states of the various proof attempts. Roughly speaking, these *proof dependencies* arise from reductions of goals to subgoals and from applications of goals to other goals as induction hypotheses or as lemmas. It will become evident in Chapter 6 that by incorporating proof state graphs the proposed framework for inductive theorem proving is capable of supporting

- the delayed or *lazy* generation of induction hypotheses (see Section 2.2.3)
- *mutual induction* for conjectures about mutually recursive operators
- applications of *unproved* lemmas (i.e. conjectures without complete proofs)
- *multiple* complete or incomplete proof attempts for the same formula.

These requirements call for some form of explicit representation and management of the proof dependencies. Last but not least, we are interested in a method of *recognizing* inductive validity of formulas that is solely based on the analysis of the proof dependencies in a possibly *incomplete* proof attempt. For example, once the base case of an induction proof has been completed the formula of the goal corresponding to the base case should easily be recognized as inductively valid.

Note that an interesting approach to the adequate representation of proof constructions in the context of inductive theorem proving, which, however, does not satisfy all of the requirements listed above, is described by Kapur, Nie, and Musser (1994).

2.3 An Example of a Proof Construction

In order to achieve on the part of the reader a better understanding of the following chapters, we continue the introduction to Part I by briefly explaining the overall idea of our formalization of inductive theorem proving and by illustrating it with a simple example. The sole purpose of this section is to convey an *intuitive* impression of how one can construct proofs of inductive theorems within the proposed framework.

Let $spec_{\mathsf{plus}}$ be an admissible specification that defines the addition operation plus on the natural numbers by

$$\begin{array}{lcl} \mathsf{plus}(x, \mathsf{0}) & = & x \\ \mathsf{plus}(x, \mathsf{s}(y)) & = & \mathsf{s}(\mathsf{plus}(x, y)) \end{array}$$

where $\mathsf{0}$ and s are the usual constructors for the natural numbers. Figure 2.1 depicts a proof state graph for the inductive theorem $\mathsf{plus}(\mathsf{0}, y) = y$ w.r.t. $spec_{\mathsf{plus}}$.

The inference rules of our calculus for inductive proofs reduce goals to (possibly empty) sets of (sub-) goals. Therefore, the notion of a goal is fundamental in our formal framework for inductive theorem proving. A *goal* $\langle \Gamma \,;\, w \rangle$ consists of a clause Γ and a so-called weight w. The purpose of the *weight* w is to provide a measure of the "size" of the clause Γ. By associating weights with clauses in goals we obtain a more flexible and powerful induction order, which is needed for guaranteeing that every induction hypothesis is of a "smaller size" than the clause it is applied to.

So given a conjecture Γ, the tuple comprising the so-called *induction variables* of Γ often yields a suitable weight for Γ. Accordingly, in the case of our example conjecture, the goal to be proved is $\langle\, \mathsf{plus}(\mathsf{0}, y) = y \;;\; y \rangle$.[7]

Proving inductive theorems in our framework basically amounts to constructing proof state graphs. As can be seen in Figure 2.1, a proof state graph is a directed labeled graph. Each node in a proof state graph is either (i) an *axiom node* labeled with a goal $\langle \Pi \,;\, () \rangle$ that consists of an axiom Π and the empty tuple, (ii) a *goal node* labeled with any goal or (iii) an *inference node*, which is labeled with information describing a particular inference step (in Figure 2.1 the name of the applied inference rule). Every proof state graph initially consists of axioms nodes for the axioms of the specification and of single goal nodes for the conjectures to be proved, and it grows by applications of inference rules to goal nodes. The purpose of a proof state graph is to record the essential dependencies among the (sub-) goals in a proof construction. Such a proof dependency can arise from the reduction of a goal to subgoals with an inference rule or from the application of a goal to another goal as an induction hypothesis or as a lemma. In the latter case, the arc connecting the corresponding inference node with the node of the applied goal is labeled with $\mathcal{I}$ or $\mathcal{L}$, respectively.

For instance, in the construction of the proof state graph in Figure 2.1 there was an application of the inference rule $\mathsf{Inductive\ Rewriting}$ to the goal

$$\langle\, \mathsf{s}(\mathsf{plus}(\mathsf{0}, z)) = \mathsf{s}(z) \;;\; \mathsf{s}(z) \rangle$$

[7] We write y instead of (y).

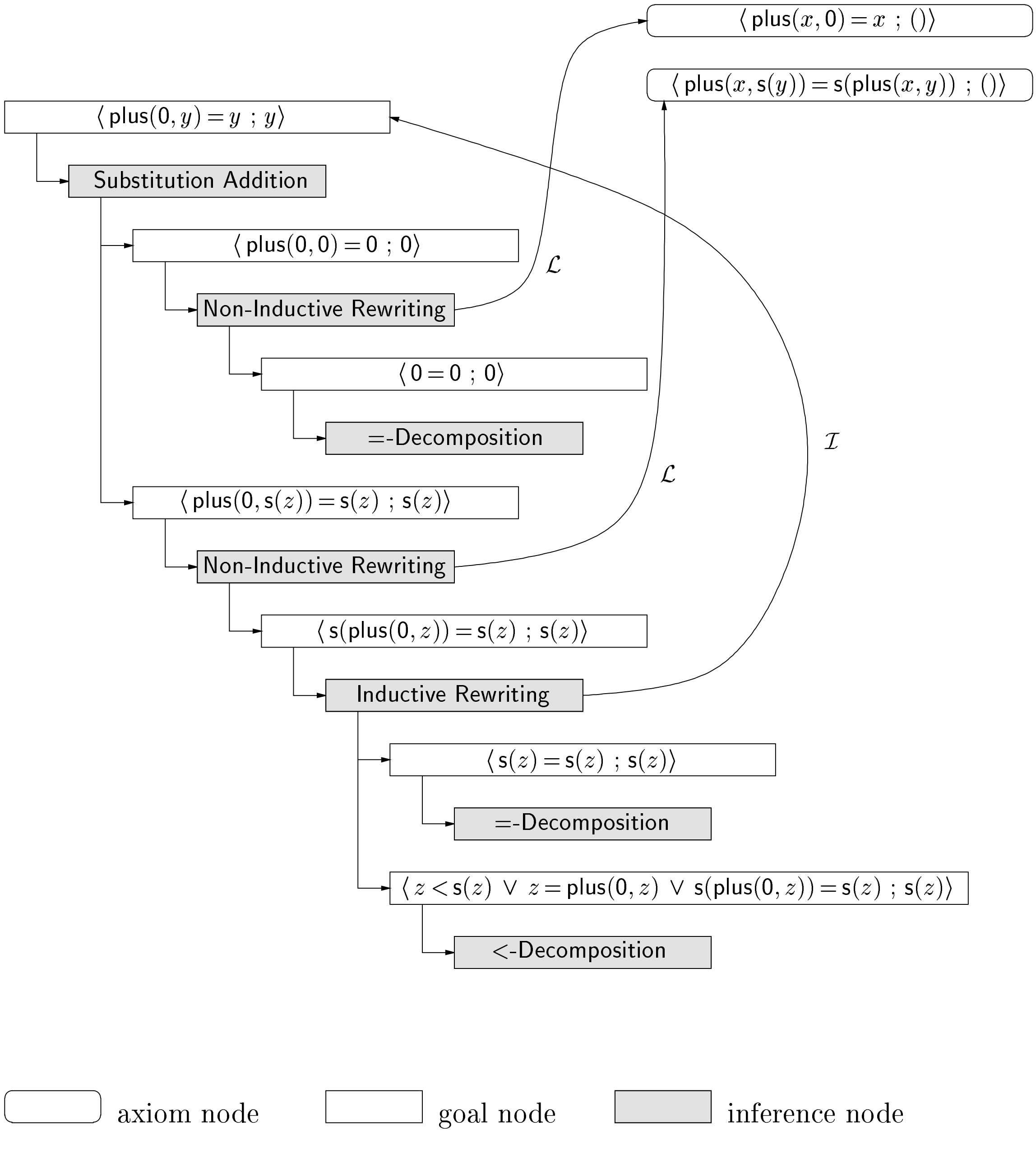


Figure 2.1: A proof (state) graph for the inductive theorem $\mathsf{plus}(0, y) = y$

that yielded the two new goals

$$\langle\, \mathsf{s}(z) = \mathsf{s}(z) \;;\; \mathsf{s}(z) \rangle \qquad \text{and} \qquad \langle\, z < \mathsf{s}(z) \,\vee\, z = \mathsf{plus}(\mathsf{0}, z) \,\vee\, \mathsf{s}(\mathsf{plus}(\mathsf{0}, z)) = \mathsf{s}(z) \;;\; \mathsf{s}(z) \rangle$$

so that a new inference node and two new goal nodes were generated. Since this inference step also involved the use of the goal $\langle\, \mathsf{plus}(\mathsf{0}, y) = y \;;\; y \rangle$ as induction hypothesis, the proof state graph contains an $\mathcal{I}$-labeled arc leading from the new inference node to the goal node for the conjecture. Note that the second of the new goals is a so-called *order subgoal* which represents the order condition that results from this application of an induction hypothesis.

So in order to obtain a proof of an inductive theorem Γ in our framework, a proof state graph needs to be constructed that contains a so-called *proof graph* for a goal of the form $\langle \Gamma \,;\, w \rangle$. Essentially, a proof graph for a goal $\langle \Gamma \,;\, w \rangle$ can be thought of as a subgraph P of the constructed proof state graph which represents a complete or "closed" proof attempt for $\langle \Gamma \,;\, w \rangle$ — closed in the sense that there are no "open" goal nodes in P and that all the lemmas "used in P" are proved. Evidently, the proof state graph in Figure 2.1 is also a proof graph for the goal $\langle\, \mathsf{plus}(\mathsf{0}, y) = y \;;\; y \rangle$.

2.4 Basic Notions and Notations

This sections contains a compilation of the basic notions and notations which will be used throughout the thesis without further explanations. We assume that the reader is familiar with fundamental concepts and the terminology of equational logic, term rewriting, algebraic specification and inductive theorem proving (for surveys see e.g. Avenhaus & Madlener, 1990; Dershowitz & Jouannaud, 1990; Wirsing, 1990; Walther, 1994).

A many-sorted *signature* $sig = (S, F, \alpha)$ comprises a set S of sort symbols, a set F of function symbols and an arity function α mapping F into S^+. For $f \in F$, $\alpha(f) = s_1 \ldots s_n s$ indicates the argument sorts $s_1 \ldots s_n$ and the result sort s of f. For every signature $sig = (S, F, \alpha)$, we assume a fixed S-sorted family of mutually disjoint sets of *variables* $V = (V_s)_{s \in S}$ where $F \cap V = \emptyset$. The well-sorted *terms* (over sig and V) are denoted by $\mathcal{T}(sig, V) = (\mathcal{T}(sig, V)_s)_{s \in S}$, and $\mathcal{GT}(sig) = (\mathcal{GT}(sig)_s)_{s \in S}$ is used for the ground terms (over sig).

We write $\mathrm{Var}(t)$ to refer to the set of variables in a term t. The length $|t|$ of a term is defined by $|x| = 1$ and $|f(t_1, \ldots, t_n)| = 1 + |t_1| + \ldots + |t_n|$. For $x \in V$, $|t|_x$ stands for the number of occurrences of x in t. A term t is called *linear* if $|t|_x = 1$ for each $x \in \mathrm{Var}(t)$. We use $\mathrm{top}(t)$ for the leading function or variable symbol in t, i.e. $\mathrm{top}(x) = x$ and $\mathrm{top}(f(t_1, \ldots, t_n)) = f$. A *position* p within a term t is a sequence of positive integers. ε is used for the root position. By t/p, we denote the sub-term of t at position p, and $t[u]_p$ means the term t with its sub-term t/p replaced with a term u. We use $\mathrm{Pos}(t)$ for the set of all positions of t. A position p is said to be *minimal* in a subset $P \subseteq \mathrm{Pos}(t)$ if $p \in P$ and there is no $p' \in P$ such that p' is a proper prefix of p.

A *substitution* is a function $\sigma\colon V \to \mathcal{T}(sig, V)$ where $\sigma(x) \in \mathcal{T}(sig, V)_s$ for all $x \in V_s$. The unique extension of σ to a function on $\mathcal{T}(sig, V)$ is also denoted by σ. We use postfix notation to denote applications of substitutions to terms and say that $t\sigma$ is an (σ-) *instance* of t.

A *sig-algebra* $\mathcal{A} = (A, F^{\mathcal{A}})$ is given by $A = (A_s)_{s\in S}$ and $F^{\mathcal{A}} = (f^{\mathcal{A}})_{f\in F}$ where

(a) for all $s \in S$, A_s is a non-empty set called the carrier of $\mathcal{A}$ for s; and

(b) for all $f \in F$ with $\alpha(f) = s_1 \ldots s_n s$, $f^{\mathcal{A}}\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$ is a function.

Let $\mathcal{B} = (B, F^{\mathcal{B}})$ be another *sig*-algebra. A *sig-homomorphism* $h\colon \mathcal{A} \to \mathcal{B}$ is a family $h = (h_s)_{s\in S}$ of functions $h_s\colon A_s \to B_s$ such that for all $f \in F$ and for all $a_i \in A_{s_i}$

$$h_s(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$$

where $\alpha(f) = s_1 \ldots s_n s$. If all functions $h_s\colon A_s \to B_s$ are bijective (surjective), then h is called a *sig-isomorphism* (*sig-epimorphism*).

A *sig*-algebra $\mathcal{I}$ is *initial* in a class K of *sig*-algebras if $\mathcal{I} \in K$ and for every $\mathcal{A} \in K$ there is a unique *sig*-homomorphism from $\mathcal{I}$ to $\mathcal{A}$. Observe that the ground term algebra $\mathcal{GT}(sig)$ is initial in the class of all *sig*-algebras. By $\text{eval}^{\mathcal{A}}$, we denote the unique *sig*-homomorphism from $\mathcal{GT}(sig)$ to a *sig*-algebra $\mathcal{A}$ defined by

$$\text{eval}^{\mathcal{A}}(f(t_1, \ldots, t_n)) = f^{\mathcal{A}}(eval^{\mathcal{A}}(t_1), \ldots, eval^{\mathcal{A}}(t_n))$$

for all $f \in F$ and for all $t_i \in \mathcal{GT}(sig)_{s_i}$. We usually write $t^{\mathcal{A}}$ instead of $\text{eval}^{\mathcal{A}}(t)$.

A *sig-congruence* on $\mathcal{A}$ is a family $\sim\, = (\sim_s)_{s\in S}$ of equivalences $\sim_s$ on A_s such that $a_i \sim_{s_i} b_i$ for $i = 1, \ldots, n$ implies $f^{\mathcal{A}}(a_1, \ldots, a_n) \sim_s f^{\mathcal{A}}(b_1, \ldots, b_n)$ for all $f \in F$. Every *sig*-homomorphism $h\colon \mathcal{A} \to \mathcal{B}$ induces a *sig*-congruence on $\mathcal{A}$, namely the *kernel* $\ker(h)$ of h defined by $\ker(h)_s = \{\, (a, b) \mid a, b \in A_s \text{ and } h_s(a) = h_s(b) \,\}$ for all $s \in S$. The *quotient algebra* $\mathcal{A}/\!\sim$ of $\mathcal{A}$ modulo a *sig*-congruence $\sim$ is the *sig*-algebra $\mathcal{Q} = (Q, F^{\mathcal{Q}})$ satisfying

(a) for all $s \in S$, $Q_s = \{\, [a] \mid a \in A_s \,\}$ where $[a] = \{\, b \in A_s \mid a \sim b \,\}$; and

(b) for all $f \in F$ and for all $a_i \in A_{s_i}$, $f^{\mathcal{Q}}([a_1], \ldots, [a_n]) = [f^{\mathcal{A}}(a_1, \ldots, a_n)]$.

Let A be a set and $\longrightarrow$ be a (binary) relation on A. Then $\longleftarrow$ is its reverse, $\longleftrightarrow$ its symmetric closure, $\xrightarrow{+}$ its transitive closure and $\xrightarrow{*}$ its transitive-reflexive closure. By $\downarrow$, we denote the *joinability* relation $\xrightarrow{*} \circ \xleftarrow{*}$ of $\longrightarrow$, that is $a_1 \downarrow a_2$ if there is an a such that $a_1 \xrightarrow{*} a \xleftarrow{*} a_2$. If $\xleftarrow{*} \circ \xrightarrow{*} \subseteq \downarrow$ then $\longrightarrow$ is *confluent*. Moreover, $\longrightarrow$ is called *terminating* if there is no infinite sequence $(a_i)_{i\in\mathbb{N}}$ in A such that $a_i \longrightarrow a_{i+1}$ for all $i \in \mathbb{N}$. An element a is $\longrightarrow$*-irreducible* if there is no a' such that $a \longrightarrow a'$.

A relation $\lesssim$ on A is said to be a *quasi-order* on A if $\lesssim$ is reflexive on A and transitive. Its *strict part* $<$ is defined by $< \,=\, \lesssim \setminus \gtrsim$ and its *equivalence* is given by $\approx \,=\, \lesssim \cap \gtrsim$. A quasi-order $\leq$ on A is called a *partial order* on A if $\leq$ is also antisymmetric. It is easily seen that the strict part $<$ of a quasi-order (or a partial order) is an irreflexive and transitive relation. If $>$ is terminating then $\lesssim$ (or $\leq$) is called *well-founded.*

We use A^* for the set of n-tuples ($n \in \mathbb{N}$) or *finite sequences* of elements in A. By $|w|$, we denote the length of the sequence $w \in A^*$. Given a partial order $\leq$ on A, we can extend $\leq$ to a partial order $\leq^{\text{lex}}$ on A^* (the *lexicographical order* induced by $\leq$) as follows. First we define its strict part $<^{\text{lex}}$ on A^* by $a_1 \dots a_m <^{\text{lex}} b_1 \dots b_n$ if

(a) $m = 0$ and $n > 0$; or

(b) $m, n > 0$ and $a_1 < b_1$; or

(c) $m, n > 0$, $a_1 = b_1$ and $a_2 \dots a_m <^{\text{lex}} b_2 \dots b_n$.

Now let $\leq^{\text{lex}}$ be the reflexive closure of $<^{\text{lex}}$ on A^*. Then $\leq^{\text{lex}}$ is a partial order on A^* such that $a_1 \leq a_2$ entails $a_1 \leq^{\text{lex}} a_2$ for all $a_1, a_2 \in A$. Note that the extension of $\leq$ to $\leq^{\text{lex}}$ does not preserve well-foundedness: $a_1 < a_2$ for some $a_1, a_2 \in A$ gives rise to the infinite descending chain $a_2 >^{\text{lex}} a_1 a_2 >^{\text{lex}} a_1 a_1 a_2 >^{\text{lex}} \dots$ However, well-foundedness of $\leq$ implies that $\leq^{\text{lex}}$ is well-founded on the set of sequences *not longer than* k, i.e. on $\{\, w \in A^* \mid |w| \leq k \,\}$, where k is any (fixed) natural number (see Wechler, 1992; Avenhaus, 1995).

Let $X \subseteq V$ be an S-sorted family of variables and $\longrightarrow$ be a relation on $\mathcal{T}(sig, X)$. We call $\longrightarrow$ *monotonic* if $t_1 \longrightarrow t_2$ implies $t[t_1]_p \longrightarrow t[t_2]_p$ for all $t_1, t_2, t \in \mathcal{T}(sig, X)$ and for all $p \in \text{Pos}(t)$ where $t_1, t_2, t/p \in \mathcal{T}(sig, X)_s$ for some $s \in S$. We speak of *sort-invariance* of $\longrightarrow$ if $t_1 \longrightarrow t_2$ entails that $t_1, t_2 \in \mathcal{T}(sig, X)_s$ for some $s \in S$. Observe that $\xleftrightarrow{*}$ is a *sig*-congruence on $\mathcal{T}(sig, X)$ if $\longrightarrow$ is monotonic and sort-invariant.

Chapter 3

The Specification Language

In this chapter we present the algebraic (i.e. equational) specification language that forms the basis of the proposed formal framework for inductive theorem proving. Our specification language can be described by its syntax, its (inductive) semantics and its admissibility conditions. While the syntax determines the signature and the set of axioms admitted in a specification, the semantics indicates what model class is to be associated with a specification as its meaning. Furthermore, the admissibility conditions have to guarantee that the semantics is actually meaningful, i.e. the associated model class is not empty. Roughly speaking, our specification language differs from many other specification formalisms for inductive theorem proving in that it admits positive/*negative* conditional equations as axioms and in that it is particularly suited to the formalization of data types with *partial* operations (see also Kühler & Wirth, 1997). For our requirements with regard to the specification language we refer to Section 2.2.1, which also contains the overall motivation for this chapter.

This chapter is organized as follows: Section 3.1 deals with the syntax and model semantics of the so-called specifications with constructors. Besides, the notion of a data model is introduced which serves as a basis for a suitable (total) algebra semantics. In 3.2 we define the admissibility conditions of our specification language. For this purpose, we associate a rewrite relation with every specification with constructors. Moreover, we show that every admissible specification has a distinguished data model with algebraic properties which resemble those of the initial model of a usual equational specification. Since admissibility of a specification requires confluence of the associated rewrite relation, we provide an easily verifiable confluence criterion in 3.3 that does not presuppose termination of the rewrite relation. In 3.4 finally, we define an appropriate notion of inductive validity with respect to an admissible specification.

3.1 Specifications with Constructors

Our interest in inductive theorem proving is mainly due to its fundamental significance to methods that allow formal reasoning about *data types*. Formal proofs of statements which express valid properties of the operations of a given data type require a preceding formalization of the data type. To state more precisely what we mean by a "data type" we quote the following conceptual definition by Ehrig and Mahr (1985):

> "A data type is a collection of data domains, designated basic data items, and operations on these domains such that all data items of the data domains can be generated from the basic data items by use of the operations. Moreover the data domains are assumed to be countable."

It is generally accepted that *initial algebra semantics* for usual (positive conditional) equational specifications is rarely appropriate when data types with partial or non-terminating operations have to be formalized (see Wirsing, 1990; Wirth & Gramlich, 1994b). Consider e.g. the specification whose axioms are those from Example 2.2.1 that define minus and less. The carrier of its initial model for the sort bool consists of *infinitely many* elements represented by "junk terms" like $\mathsf{less}(0, \mathsf{minus}(0, \mathsf{s}^{n+1}(0)))$ instead of just *two* (for true and false). In some cases, lack of sufficient completeness as the essential problem can be avoided by demanding that the specification describe total extensions of the partial operations. For our specifications with constructors, however, we do not require sufficient completeness for adequate representations of data types with partial operations any more — mainly because we acknowledge the importance constructors have in describing the data items of a data type.

3.1.1 Syntax of Specifications with Constructors

In order to ensure that a data type D with partial operations will be adequately represented by certain model classes of a specification of D (see below), our specification language requires the user to indicate the constructors for D, i.e. those function symbols which are needed for designating the data items of D. Let $sig = (S, F, \alpha)$ be a signature. Formally, a subset $C \subseteq F$ is said to be a set of *constructors* for sig if the signature $sig^C = (S, C, \alpha|_C)$ induced by C is *sensible*, i.e. for each $s \in S$ there is at least one constructor ground term $t \in \mathcal{GT}(sig^C)_s$ of sort s. We call sig^C the *constructor signature* of sig.

We assume that for each sort $s \in S$, the set V_s of variables for s is composed of two disjoint subsets V_s^{C} and V_s^{G}. The elements of V^{C} are called *constructor variables*, while the elements of V^{G} are called *general variables*. Intuitively, constructor variables range over data items only, whereas general variables allow statements about undefined objects as well. $\mathcal{T}(sig^C, V^{\mathrm{C}})$ denotes the set of (pure) *constructor terms*. A substitution σ: $V \to \mathcal{T}(sig, V)$ is said to be a *constructor substitution* if $\sigma(V^{\mathrm{C}}) \subseteq \mathcal{T}(sig^C, V^{\mathrm{C}})$, and we call σ an *inductive substitution* if $\sigma(V^{\mathrm{C}}) \subseteq \mathcal{GT}(sig^C)$ and $\sigma(V^{\mathrm{G}}) \subseteq \mathcal{T}(sig, V^{\mathrm{G}})$.

An *equation* is a pair $t_1 = t_2$ such that $t_1, t_2 \in \mathcal{T}(sig, V)_s$ for some $s \in S$. An *atom* is an equation or a *definedness* atom $\mathrm{def}(t)$ where 'def' is a predefined predicate symbol and $t \in \mathcal{T}(sig, V)$. Informally, $\mathrm{def}(t)$ means that the applications of operations denoted by t can be evaluated to data items. A *positive literal* is an atom, and a *negative literal* is a negated atom. A *literal* is a positive or a negative literal. The *complement* $\overline{\lambda}$ of a positive literal λ is $\neg\lambda$, and the complement $\overline{\neg\lambda}$ of a negative literal $\neg\lambda$ is λ. A (disjunctive) *clause* is a possibly empty sequence $\lambda_1 \dots \lambda_n$ of literals.

A (positive) *conditional equation* is an expression of the form $l = r \leftarrow \Delta$ where Δ is a possibly empty sequence of (positive) *condition literals*. The clause representation of $l = r \leftarrow \lambda_1 \dots \lambda_n$ is $(l = r)\, \overline{\lambda_1} \dots \overline{\lambda_n}$. When emphasizing its directed use we call a conditional equation a (conditional) *rewrite rule*.

Given expressions $e_1, \dots, e_n$ (e.g. terms, clauses or a rewrite rules), let $\mathrm{Var}(e_1, \dots, e_n)$ denote the set of variables occurring in any of $e_1, \dots, e_n$.

Definition 3.1.1 A *specification with constructors* $spec = (sig, C, E)$ is composed of a signature sig such that C is a set of constructors for sig, and of a set E of conditional equations (over sig and V).

Note that at this point we do not place any restrictions on the conditional equations to be used in specifications. In Section 3.2, however, we will develop restricting admissibility conditions which will guarantee that appropriate notions of inductive semantics are actually meaningful for admissible specifications.

3.1.2 Model Semantics of Specifications with Constructors

We now have to determine the elements of a sig-algebra $\mathcal{A}$ which may be assigned to constructor variables. As constructor variables are meant to range over data items only, we assign those elements of $\mathcal{A}$ which are designated by constructor ground terms to constructor variables. These elements form the carriers of a sig^C-algebra, which we call the *data reduct* $\mathcal{A}^C$ of $\mathcal{A}$:

Definition 3.1.2 Let $\mathcal{A} = (A, F^{\mathcal{A}})$ be a sig-algebra. The *data reduct* of $\mathcal{A}$ is the sig^C-algebra $\mathcal{A}^C = (A^C, C^{\mathcal{A}^C})$ satisfying

(a) for each $s \in S$, $A^C_s = \{\, t^{\mathcal{A}} \in A_s \mid t \in \mathcal{GT}(sig^C)_s \,\}$; and

(b) for each $c \in C$ and for all $a_i \in A^C_{s_i}$, $c^{\mathcal{A}^C}(a_1, \dots, a_n) = c^{\mathcal{A}}(a_1, \dots, a_n)$ where $\alpha(c) = s_1 \dots s_n s$.

The data reduct $\mathcal{A}^C$ is in fact a sig^C-algebra: Since C is a set of constructors for sig, sig^C is sensible, and so $A^C_s \neq \emptyset$ for each $s \in S$. Furthermore, let $a_1, \dots, a_n \in A^C$. Then there are $t_1, \dots, t_n \in \mathcal{GT}(sig^C)$ such that $c^{\mathcal{A}}(a_1, \dots, a_n) = c^{\mathcal{A}}(t_1^{\mathcal{A}}, \dots, t_n^{\mathcal{A}})$, and since $\mathrm{eval}^{\mathcal{A}}\colon \mathcal{GT}(sig) \to \mathcal{A}$ is a sig-homomorphism, we have $c^{\mathcal{A}}(t_1^{\mathcal{A}}, \dots, t_n^{\mathcal{A}}) = c(t_1, \dots, t_n)^{\mathcal{A}}$. Hence, $c^{\mathcal{A}}(a_1, \dots, a_n) \in A^C$, which shows that A^C is closed under $c^{\mathcal{A}}$ for each $c \in C$.[1]

Given an S-sorted family of variables $X \subseteq V$, the data reduct of the term algebra $\mathcal{T}(sig, X)$ is obviously $\mathcal{GT}(sig^C)$. Moreover, the image of a data reduct under a sig-homomorphism is a data reduct:

[1] $\mathcal{A}^C$ is the (sig^C-) homomorphic image of $\mathcal{GT}(sig^C)$ under $\mathrm{eval}^{\mathcal{A}}$, and hence a term-generated sub-algebra of the sig^C-reduct $\mathcal{A}|_{sig^C}$ of $\mathcal{A}$.

Lemma 3.1.3 *Let $\mathcal{A}$ and $\mathcal{B}$ be sig-algebras, and let $h\colon \mathcal{A} \to \mathcal{B}$ be a sig-homomorphism. Let $h^C = (h^C_s)_{s\in S}$ be the family of functions defined by $h^C_s = h_s|_{A^C_s}$ for all $s \in S$. Then $h^C\colon \mathcal{A}^C \to \mathcal{B}^C$ is a sig^C-epimorphism.*

We can now give meaning to terms, literals and clauses.

Definition 3.1.4 Let $\mathcal{A} = (A, F^{\mathcal{A}})$ be a *sig*-algebra.

(i) Let $X \subseteq V$. A *valuation* of X in $\mathcal{A}$ is a function $\varphi\colon X \to \mathcal{A}$ such that $\varphi(x) \in A^C_s$ for every $x \in X_s \cap V^{\mathrm{C}}_s$ and $\varphi(x) \in A_s$ for every $x \in X_s \cap V^{\mathrm{G}}_s$. By $\mathrm{eval}^{\mathcal{A}}_{\varphi}$ we denote the unique *sig*-homomorphism from $\mathcal{T}(sig, X)$ to $\mathcal{A}$ that extends φ.

(ii) Let φ be a valuation of V in $\mathcal{A}$. Then $\mathcal{A}$ *satisfies* an equation $t_1 = t_2$ *with* φ if $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t_1) = \mathrm{eval}^{\mathcal{A}}_{\varphi}(t_2)$, and $\mathcal{A}$ satisfies a definedness atom $\mathrm{def}(t)$ for $t \in \mathcal{T}(sig, V)_s$ with φ if $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t) \in A^C_s$. Moreover, $\mathcal{A}$ 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 a literal in Γ which $\mathcal{A}$ satisfies with φ.

(iii) A clause Γ is *valid* in $\mathcal{A}$ if $\mathcal{A}$ satisfies Γ with every valuation of V in $\mathcal{A}$. This is denoted by $\mathcal{A} \models \Gamma$. Let K be a class of *sig*-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.

The inductive substitutions are exactly the valuations of V in $\mathcal{T}(sig, V^{\mathrm{G}})$. Besides the *equality axioms* of a signature *sig*, there are other clauses which are valid in all *sig*-algebras, e.g. $\mathrm{def}(c(X_1, \ldots, X_n)) \vee \neg\mathrm{def}(X_1) \vee \ldots \vee \neg\mathrm{def}(X_n)$ where $c \in C$ is a constructor and $X_i \in V^{\mathrm{G}}_{s_i}$ for $i = 1, \ldots, n$.

The following useful result relates valuations and constructor substitutions.

Lemma 3.1.5 *Let $\mathcal{A}$ be a sig-algebra, $X \subseteq V$, φ be a valuation of X in $\mathcal{A}$ and σ be a constructor substitution with $\sigma(V) \subseteq \mathcal{T}(sig, X)$. Then $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t\sigma) = \mathrm{eval}^{\mathcal{A}}_{(\mathrm{eval}^{\mathcal{A}}_{\varphi} \circ \sigma)}(t)$ for all $t \in \mathcal{T}(sig, V)$.*

The models of a specification with constructors can now be defined as usual.

Definition 3.1.6 Let $spec = (sig, C, E)$ be a specification with constructors. A *sig*-algebra $\mathcal{A}$ is called a (*sig*-) *model* of *spec* if (the clause representation of) each conditional equation in E is valid in $\mathcal{A}$. The class of all *sig*-models of *spec* is denoted by $\mathrm{Mod}(spec)$.

Since the clause representation of any conditional equation contains at least one positive literal, every conditional equation (over *sig* and V) is valid in the trivial *sig*-algebra whose carriers consist of one element each. Therefore, $\mathrm{Mod}(spec)$ is not empty for any specification with constructors *spec*. Still, we do not really consider $\mathrm{Mod}(spec)$ an appropriate inductive semantics for a specification with constructors *spec*. The reason for that is that $\mathrm{Mod}(spec)$ includes also those *sig*-models of *spec* which unnecessarily equate or *confuse* data items (see below). In other words, *sig*-models of this kind satisfy equations between constructor ground terms that are not valid in all *sig*-models of *spec*. An extreme example of such a *sig*-model is the trivial *sig*-algebra.

3.1.3 Data Models

In eliminating those models from $\mathrm{Mod}(\mathit{spec})$ that confuse data items we obtain a particularly suited class of models as the semantics for a specification with constructors. Since usually the data reduct of each of the models in the resulting class yields a one-to-one representation of the data domains of the given data type we call these models *data models*.

Definition 3.1.7 Let $\mathit{spec} = (\mathit{sig}, C, E)$ be a specification with constructors. We say that a sig-model $\mathcal{A}$ of spec is a *data model* of spec if, for all constructor ground terms $t_1, t_2 \in \mathcal{GT}(\mathit{sig}^C)$, $t_1^{\mathcal{A}} = t_2^{\mathcal{A}}$ implies $\mathrm{Mod}(\mathit{spec}) \models t_1 = t_2$. Let $\mathrm{DMod}(\mathit{spec})$ denote the class of all data models of spec.

Note that $\mathrm{DMod}(\mathit{spec})$ may be empty (see below). Moreover, the data reducts of any two data models in $\mathrm{DMod}(\mathit{spec})$ are isomorphic:

Lemma 3.1.8 *Let $\mathcal{A}$ be a sig-model of spec. Then $\mathcal{A}$ is a data model of spec if and only if its data reduct $\mathcal{A}^C$ is initial in the class of sig^C-algebras $\{\, \mathcal{B}^C \mid \mathcal{B} \in \mathit{Mod}(\mathit{spec}) \,\}$.*

Corollary 3.1.9 *Let $\mathcal{A}$ and $\mathcal{B}$ be data models of spec. Then their data reducts $\mathcal{A}^C$ and $\mathcal{B}^C$ are (sig^C-) isomorphic.*

Consequently, data models do not differ in the evaluation of constructor terms, i.e. for any $\mathcal{A}, \mathcal{B} \in \mathrm{DMod}(\mathit{spec})$ and $t_1, t_2 \in \mathcal{T}(\mathit{sig}^C, V^C)$ we have $\mathcal{A} \models t_1 = t_2$ iff $\mathcal{B} \models t_1 = t_2$. For general terms, however, the corresponding statement does not hold — not even for ground terms whose definedness is valid in all models of spec:

Example 3.1.10 Let $\mathit{spec} = (\mathit{sig}, C, E)$ be the specification with constructors over the signature $\mathit{sig} = (S, F, \alpha)$ such that $S = \{\mathsf{any}\}$, $C = \{\mathsf{c}_1, \mathsf{c}_2\}$, $F = C \cup \{\mathsf{d}\}$, $\alpha(o) = \mathsf{any}$ for each $o \in F$, and $E = \{\, \mathsf{d} = \mathsf{c}_1 \leftarrow \mathsf{d} \neq \mathsf{c}_2 \,\}$.

Let $\mathcal{A} = (A, F^{\mathcal{A}})$ be the sig-algebra with $A_{\mathsf{any}} = \{a_1, a_2\}$, $\mathsf{c}_1^{\mathcal{A}} = a_1$, $\mathsf{c}_2^{\mathcal{A}} = a_2$ and $\mathsf{d}^{\mathcal{A}} = a_1$. Since $\mathsf{c}_1^{\mathcal{A}} \neq \mathsf{c}_2^{\mathcal{A}}$ and $\mathsf{d}^{\mathcal{A}} = \mathsf{c}_1^{\mathcal{A}}$ we have $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$. Moreover, let $\mathcal{B} = (B, F^{\mathcal{B}})$ be the sig-algebra with $B_{\mathsf{any}} = \{a_1, a_2\}$, $\mathsf{c}_1^{\mathcal{B}} = a_1$, $\mathsf{c}_2^{\mathcal{B}} = a_2$ and $\mathsf{d}^{\mathcal{B}} = a_2$. Now $\mathsf{c}_1^{\mathcal{B}} \neq \mathsf{c}_2^{\mathcal{B}}$ and $\mathsf{d}^{\mathcal{B}} = \mathsf{c}_2^{\mathcal{B}}$, so $\mathcal{B}$ is also a data model of spec.

Hence, we have $\mathcal{A}, \mathcal{B} \in \mathrm{DMod}(\mathit{spec})$, $\mathrm{Mod}(\mathit{spec}) \models \mathrm{def}(\mathsf{d})$ and $\mathsf{d}^{\mathcal{A}} = \mathsf{c}_1^{\mathcal{A}}$, but *not* $\mathsf{d}^{\mathcal{B}} = \mathsf{c}_1^{\mathcal{B}}$.

In Section 3.2, however, we will show that all "defined" terms are uniformly evaluated in all data models of *admissible* specifications (see Theorem 3.2.8).

We have mentioned before that the class of all data models of a specification with constructors may be empty. In order to demonstrate this and to motivate our admissibility conditions for guaranteeing the existence of data models (see Section 3.2), we give the following two examples.

Example 3.1.11 Let $sig = (S, F, \alpha)$ be the signature with $S = \{\mathsf{any}\}$, $C = \{\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3\}$, $F = C \cup \{\mathsf{d}\}$, and $\alpha(o) = \mathsf{any}$ for each $o \in F$. Let $E_1 = \{\, \mathsf{c}_1 = \mathsf{c}_2 \leftarrow \mathsf{c}_1 \neq \mathsf{c}_3 \,\}$ and $spec_1 = (sig, C, E_1)$.

Obviously, neither $\mathrm{Mod}(spec_1) \models \mathsf{c}_1 = \mathsf{c}_2$ holds nor $\mathrm{Mod}(spec_1) \models \mathsf{c}_1 = \mathsf{c}_3$, but for any $\mathcal{A} \in \mathrm{Mod}(spec_1)$ we have $\mathsf{c}_1^{\mathcal{A}} = \mathsf{c}_2^{\mathcal{A}}$ or $\mathsf{c}_1^{\mathcal{A}} = \mathsf{c}_3^{\mathcal{A}}$ (because of E_1). Hence $\mathcal{A}$ cannot be a data model of $spec_1$. This shows that $\mathrm{DMod}(spec_1) = \emptyset$.

Example 3.1.12 Let sig and C be as in the preceding example, $E_2 = \{\, \mathsf{d} = \mathsf{c}_1 \leftarrow \mathsf{c}_1 \neq \mathsf{c}_3,\ \mathsf{d} = \mathsf{c}_2 \leftarrow \mathsf{c}_1 \neq \mathsf{c}_3 \,\}$, and let $spec_2 = (sig, C, E_2)$.

One easily shows that neither $\mathrm{Mod}(spec_2) \models \mathsf{c}_1 = \mathsf{c}_3$ nor $\mathrm{Mod}(spec_2) \models \mathsf{c}_1 = \mathsf{c}_2$. Let $\mathcal{A} \in \mathrm{Mod}(spec_2)$. If $\mathsf{c}_1^{\mathcal{A}} = \mathsf{c}_3^{\mathcal{A}}$ then $\mathcal{A}$ is not a data model of $spec_2$. Otherwise, due to E_2, we have $\mathsf{c}_1^{\mathcal{A}} = \mathsf{d}^{\mathcal{A}} = \mathsf{c}_2^{\mathcal{A}}$, so $\mathcal{A}$ is not a data model of $spec_2$ either.

3.2 Admissibility Conditions

So far, we have not placed any restrictions on the set of positive/negative conditional equations in a specification with constructors $spec = (sig, C, E)$. The two examples above, however, show that certain restrictions are necessary to ensure that $\mathrm{DMod}(spec)$ is meaningful (i.e. not empty) as an inductive semantics. In this section we therefore present the admissibility conditions of our specification language which guarantee the existence of a distinguished data model $\mathcal{M}(spec)$ for any admissible specification $spec$. Hence, $\mathrm{DMod}(spec) \neq \emptyset$.

Our admissibility conditions are based on terminology and concepts from the theory of term rewriting; recall that every conditional equation can be regarded as a (conditional) rewrite rule. We will show in the following that the rewrite relation $\longrightarrow_R$ associated with an admissible specification $spec = (sig, C, R)$ can be defined in such a way that $\longrightarrow_R$ yields a sound and complete operationalization of equality in all data models: $t_1 \overset{*}{\longleftrightarrow}_R t_2$ iff $\mathrm{DMod}(spec) \models t_1 = t_2$, for $t_1, t_2 \in \mathcal{T}(sig, V^{\mathrm{G}})$. Moreover, admissibility of $spec$ will be proved to ensure that the sig-algebra $\mathcal{T}(sig, V^{\mathrm{G}})/\overset{*}{\longleftrightarrow}_R$ is a data model of $spec$, namely the so-called *standard data model* $\mathcal{M}(spec)$, which is free over V^{G} in $\mathrm{DMod}(spec)$.

3.2.1 Positive/Negative Conditional Rewrite Specifications

We begin with the idea of extending the distinction made between constructors and the other function symbols in signatures to the axioms in specifications. That is, we require that the set R of rewrite rules (or conditional equations) in a specification with constructors can be partitioned into a set R^C of *constructor rules* and a set R^D of *defining rules*. Intuitively, the constructor rules in R^C are to specify the relations on the constructor ground terms necessary for representing the data items of the given data type, while the defining rules in R^D describe the effects of the other operations of the data type *consistently* (see below).

Note that we have to forbid negative equational condition literals in constructor rules, as is shown in Example 3.1.11: The class of data models of a specification including a constructor rule with a negative equational condition may be empty, since the class of the data reducts of all models of such a specification need not contain an initial element (see Lemma 3.1.8). In defining rules, however, we do admit negative equational conditions. To prevent a defining rule $l = r \leftarrow \Delta$ from being applied to a constructor (ground) term, its left-hand side l must contain at least one non-constructor function symbol.

Definition 3.2.1 Let $sig = (S, F, \alpha)$ be a signature such that $C \subseteq F$ is a set of constructors for sig.

(i) A *constructor rule* is a rewrite rule $l = r \leftarrow u_1 = v_1, \ldots, u_n = v_n$ such that

(a) $l, r \in \mathcal{T}(sig^C, V)$ and $u_i, v_i \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ for $i = 1, \ldots, n$

(b) $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$ and $\mathrm{Var}(u_i), \mathrm{Var}(v_i) \subseteq \mathrm{Var}(l)$ for $i = 1, \ldots, n$

(ii) A *defining rule* is a rewrite rule $l = r \leftarrow \Delta$ satisfying

(a) $l \in \mathcal{T}(sig, V) \backslash \mathcal{T}(sig^C, V)$

(b) Δ does not contain any literal of the form $\neg\mathrm{def}(t)$.

A major purpose of the rewrite relation $\longrightarrow_R$ associated with a specification with constructors $spec = (sig, C, R)$ is to give an explicit characterization of the "best" data model $\mathcal{M}(spec)$ as the quotient algebra $\mathcal{T}/\overset{*}{\longleftrightarrow}_R$ of a suitable term algebra $\mathcal{T}$. It is possible to define $\longrightarrow_R$ on $\mathcal{GT}(sig)$ in such a way that admissibility of $spec$ implies initiality of $\mathcal{GT}(sig)/\overset{*}{\longleftrightarrow}_R$ in $\mathrm{DMod}(spec)$, as is done by Avenhaus and Madlener (1997) or Wirth and Gramlich (1994a). Here, however, we will define $\longrightarrow_R$ on $\mathcal{T}(sig, V^{\mathrm{G}})$, since for $\mathcal{T} := \mathcal{GT}(sig)$ a respective formulation of Theorem 3.2.8 does not hold for clauses with general variables.

In accordance with the distinction we make between the constructor rules R^C and the defining rules R^D in R, we define the rewrite relation $\longrightarrow_R$ in two steps. Firstly, rewriting constructor ground (sub-) terms is only possible with the rewrite relation $\longrightarrow_{R^C}$ induced by R^C. Since R^C is a positive conditional rewrite system $\longrightarrow_{R^C}$ can be defined as usual. Secondly, for a rewrite step $t[l\sigma]_p \longrightarrow_R t[r\sigma]_p$ with a defining rule $l = r \leftarrow \Delta$ in R^D, each condition literal in $\Delta\sigma$ must be fulfilled. This means for a negative condition $u \neq v$ in Δ that both $u\sigma$ and $v\sigma$ can be rewritten (using $\longrightarrow_R$) to constructor ground terms which are not joinable using $\longrightarrow_{R^C}$. A definedness atom $\mathrm{def}(u)$ in Δ is fulfilled if $u\sigma \overset{*}{\longrightarrow}_R \hat{u}$ for some constructor ground term $\hat{u}$.

Definition 3.2.2 Let $spec = (sig, C, R)$ be a specification with constructors and $R = R^C \uplus R^D$ where R^C is a set of constructor rules and R^D a set of defining rules.

(i) Let the sequence $(\longrightarrow_{R^C,i})_{i \in \mathbb{N}}$ of relations on $\mathcal{T}(sig, V^{\mathrm{G}})$ be defined by

(a) $\longrightarrow_{R^C,0} = \emptyset$.

(b) $t_1 \longrightarrow_{R^C,i+1} t_2$ if there is a rewrite rule $l = r \leftarrow u_1 = v_1, \ldots, u_n = v_n$ in R^C, a position $p \in \mathrm{Pos}(t_1)$ and an inductive substitution σ such that (1) $t_1/p = l\sigma$ (2) $t_2 = t_1[r\sigma]_p$ and (3) $u_k\sigma \downarrow_{R^C,i} v_k\sigma$ for $k = 1, \ldots, n$.

Then $\longrightarrow_{R^C} = \bigcup_{i\in\mathbb{N}} \longrightarrow_{R^C,i}$.

(ii) Let the sequence $(\longrightarrow_{R,i})_{i\in\mathbb{N}}$ of relations on $\mathcal{T}(sig, V^{\mathrm{G}})$ be defined by

(a) $\longrightarrow_{R,0} = \longrightarrow_{R^C}$.

(b) $t_1 \longrightarrow_{R,i+1} t_2$ if $t_1 \longrightarrow_{R^C} t_2$ or there is a rewrite rule $l = r \leftarrow \Delta$ in R^D, a position $p \in \mathrm{Pos}(t_1)$ and an inductive substitution σ such that (1) $t_1/p = l\sigma$ (2) $t_2 = t_1[r\sigma]_p$ (3) for each $u = v$ in Δ, $u\sigma \downarrow_{R,i} v\sigma$ (4) for each $\mathrm{def}(u)$ in Δ there is a $\hat{u} \in \mathcal{GT}(sig^C)$ such that $u\sigma \stackrel{*}{\longrightarrow}_{R,i} \hat{u}$ and (5) for each $u \neq v$ in Δ there are $\hat{u}, \hat{v} \in \mathcal{GT}(sig^C)$ such that $u\sigma \stackrel{*}{\longrightarrow}_{R,i} \hat{u}$, $v\sigma \stackrel{*}{\longrightarrow}_{R,i} \hat{v}$ and $\hat{u} \not\downarrow_{R^C} \hat{v}$.

Then $\longrightarrow_R = \bigcup_{i\in\mathbb{N}} \longrightarrow_{R,i}$.

Basic properties of $\longrightarrow_{R^C}$ and $\longrightarrow_R$ are listed in the following lemma.

Lemma 3.2.3

(1) $\longrightarrow_{R^C,i} \subseteq \longrightarrow_{R^C,i+1} \subseteq \longrightarrow_{R^C}$ *for all* $i \in \mathbb{N}$

(2) $\longrightarrow_{R^C} \subseteq \longrightarrow_{R,i} \subseteq \longrightarrow_{R,i+1} \subseteq \longrightarrow_R$ *for all* $i \in \mathbb{N}$

(3) *If* $t \stackrel{*}{\longrightarrow}_R t'$ *for* $t \in \mathcal{GT}(sig^C)$ *and* $t' \in \mathcal{T}(sig, V^{\mathrm{G}})$ *then* $t \stackrel{*}{\longrightarrow}_{R^C} t'$ *and* $t' \in \mathcal{GT}(sig^C)$.

(4) $\longrightarrow_R$ *is sort-invariant and monotonic.*

Example 3.1.12 shows that requiring R to consist of constructor rules and defining rules only is not sufficient for the existence of data models. Note that the specification in Example 3.1.12 contains an *inconsistent* definition for the function symbol d — inconsistent in the sense that there are constructor ground terms t_1 and t_2 such that $t_1 \stackrel{*}{\longleftrightarrow}_R t_2$ but not $t_1 \stackrel{*}{\longleftrightarrow}_{R^C} t_2$. Such inconsistencies cannot arise if $\longrightarrow_R$ is confluent: Then $t_1 \stackrel{*}{\longleftrightarrow}_R t_2$ entails $t_1 \downarrow_R t_2$ for $t_1, t_2 \in \mathcal{GT}(sig^C)$, and by applying Lemma 3.2.3(3) one obtains $t_1 \downarrow_{R^C} t_2$ and hence $t_1 \stackrel{*}{\longleftrightarrow}_{R^C} t_2$. To put it another way, confluence of $\longrightarrow_R$ guarantees that $spec = (sig, C, R)$ is a *consistent extension* of the "base" or constructor specification $spec^C = (sig^C, R^C)$ (see Ehrig & Mahr, 1985).

A further admissibility condition is needed to ensure that $\mathcal{T}(sig, V^{\mathrm{G}})/\stackrel{*}{\longleftrightarrow}_R$ is a (data) model of *spec* (see Example 3.1.10). In order to achieve correspondence of the model semantics with our method of testing negative condition literals in defining rules, there needs to be a definedness condition literal for each (non-constructor) term occurring in a negative condition literal.

Definition 3.2.4 A specification with constructors $spec = (sig, C, R)$ is called an *admissible* specification if *spec* satisfies the following conditions:

(a) $R = R^C \uplus R^D$ where R^C is a set of constructor rules and R^D a set of defining rules.

(b) $\longrightarrow_R$ is confluent.

(c) For each $l = r \leftarrow \Delta$ in R^D and for each $t \in \mathcal{T}(sig, V) \backslash \mathcal{T}(sig^C, V^{\mathrm{C}})$ occurring (on top-level) in a negative literal in Δ there is a literal $\mathrm{def}(t)$ in Δ.

Note that we do *not* require termination of $\longrightarrow_R$ in our admissibility conditions.

3.2.2 The Standard Data Model $\mathcal{M}(\mathit{spec})$

Having defined our admissibility conditions we can now show that, for any admissible specification $\mathit{spec} = (\mathit{sig}, C, R)$, the *sig*-algebra $\mathcal{T}(\mathit{sig}, V^{\mathrm{G}})/\overset{*}{\longleftrightarrow}_R$ is a data model of *spec* that plays a particular role in the class of all data models of *spec*: It is free over V^{G} in DMod(*spec*). Thus, $\mathcal{M}(\mathit{spec}) := \mathcal{T}(\mathit{sig}, V^{\mathrm{G}})/\overset{*}{\longleftrightarrow}_R$ has algebraic properties similar to those of the initial model of a usual equational specification in that $\mathcal{M}(\mathit{spec})$ may be regarded as a distinguished "representative" of the class of all data models (see Theorem 3.2.8). This gives rise to two appropriate kinds of semantics for specifications with constructors of data types with partial operations, namely (i) DMod(*spec*) and (ii) (the isomorphism class of) the so-called *standard data model* $\mathcal{M}(\mathit{spec})$.

Proposition 3.2.5 *Let spec* $= (\mathit{sig}, C, R)$ *be an admissible specification. Then*

(1) $\mathcal{T}(\mathit{sig}, V^{\mathrm{G}})/\overset{*}{\longleftrightarrow}_R$ *is a data model of spec.*

(2) *Let* $\mathcal{A}$ *be any data model of spec. Then its data reduct* $\mathcal{A}^C$ *and* $\mathcal{GT}(\mathit{sig}^C)/\overset{*}{\longleftrightarrow}_{R^C}$ *are* (sig^C-) *isomorphic.*

Hence, the data reduct of *every* data model of an admissible specification can be explicitly characterized in terms of $\mathcal{GT}(\mathit{sig}^C)$ and $\longrightarrow_{R^C}$.

The following lemma justifies the view that $\longrightarrow_R$ yields a sound and complete operationalization of equality in all data models of *spec*. It is of central importance in the proofs of Theorems 3.2.7 and 3.2.8.

Lemma 3.2.6 *Let spec* $= (\mathit{sig}, C, R)$ *be an admissible specification, and let* $t_1, t_2 \in \mathcal{T}(\mathit{sig}, V^{\mathrm{G}})$. *Then* $t_1 \overset{*}{\longleftrightarrow}_R t_2$ *iff* $\mathit{DMod}(\mathit{spec}) \models t_1 = t_2$.

Freeness of an algebra is usually defined as follows. Let K be a class of *sig*-algebras, and let $X \subseteq V$ be an S-sorted family of variables. A *sig*-algebra $\mathcal{F}(X)$ is *free* over X in K if $\mathcal{F}(X) \in K$ and there is a valuation u of X in $\mathcal{F}(X)$ such that for every valuation φ of X in a *sig*-algebra $\mathcal{A} \in K$ there is a unique *sig*-homomorphism h: $\mathcal{F}(X) \to \mathcal{A}$ such that the following diagram commutes, i.e. $\varphi = h \circ u$.

$$\begin{tikzcd} X \arrow[r, "\varphi"] \arrow[d, "u"'] \arrow[dr, phantom, "="] & \mathcal{A} \\ \mathcal{F}(X) \arrow[ur, "h"'] & \end{tikzcd}$$

Note that the free algebra is unique up to isomorphism: If *sig*-algebras $\mathcal{F}_1(X)$ and $\mathcal{F}_2(X)$ are free over X in K then $\mathcal{F}_1(X)$ and $\mathcal{F}_2(X)$ are isomorphic (see e.g. Ehrig & Mahr, 1985).

Theorem 3.2.7 *Let spec* $= (\mathit{sig}, C, R)$ *be an admissible specification. Then* $\mathcal{T}(\mathit{sig}, V^{\mathrm{G}})/\overset{*}{\longleftrightarrow}_R$ *is free over* V^{G} *in DMod(spec).*

We call $\mathcal{T}(sig, V^{\mathrm{G}})/\overset{*}{\longleftrightarrow}_R$ the *standard data model* of an admissible specification *spec* and use $\mathcal{M}(spec)$ to denote it. Its significance as a "representative" of DMod(*spec*) is confirmed in the following characterization of the relation between validity in $\mathcal{M}(spec)$ and validity in all data models of *spec*.

Theorem 3.2.8 *Let spec* $= (sig, C, R)$ *be an admissible specification, and let* Γ *be a clause such that* $\mathcal{M}(spec) \models \mathrm{def}(t)$ *for every (top-level) term* t *occurring in a negative literal of* Γ*. Then* $\mathcal{M}(spec) \models \Gamma$ *is sufficient for* $DMod(spec) \models \Gamma$*.*

In particular, every equation valid in $\mathcal{M}(spec)$ is valid in every data model of *spec*.

It should be noted that Theorems 3.2.7, 3.2.8 and 3.4.2 have counterparts in the more general specification approach by Wirth and Gramlich (1994a, 1994b) from which the specification language presented in (Kühler & Wirth, 1997) and this thesis originates.

3.3 A Confluence Criterion

Contrary to most other specification formalisms for inductive theorem proving, the one presented in this thesis does not require termination. However, its admissibility conditions require confluence, and since many interesting rewrite systems (i.e. sets of rewrite rules) are non-decreasing (see Dershowitz et al., 1988, for a definition of "decreasing") or even non-terminating (see Example 2.2.2), we need a confluence criterion that does not presuppose termination.

Note that several basic results on confluence of unconditional rewrite systems that are based on syntactic considerations do not hold in the conditional case. In particular, local confluence of conditional rewrite systems is not equivalent to joinability of all critical pairs. In other words, variable overlaps may be "critical" as well. This may even happen when termination is given (combined with left-linearity and normality; see Dershowitz et al., 1988, Example B, p. 36). If we do not require termination, the situation is even more complicated: There are left-linear positive conditional rewrite systems that do not have any critical pairs but lack confluence (see Dershowitz et al., 1988, Example A, p. 36)[2]. Therefore, reasonable syntactic confluence criteria for non-terminating rewrite systems need strengthened forms of joinability of critical pairs and syntactic restrictions on rewrite rules such as left-linearity and (weakened forms of) normality.

Another major problem is caused by the infinite number of substitutions that must be tested for fulfilling the conditions in critical pairs. Therefore, effectively testable conditions which guarantee the *infeasibility* of the critical pairs have practical relevance. The confluence criterion (see Theorem 3.3.2) presented in this thesis essentially makes use of the fact that critical pairs with complementary literals in the conditions are infeasible and need not be considered hence. It is not the strongest known con-

[2] This example was originally given by Bergstra and Klop (1986).

fluence criterion applicable to non-terminating constructor-based rewrite systems[3] but it is effectively testable. Moreover, it extends the class of specifications admitted for inductive theorem proving since it no longer requires termination. To our knowledge it is the strongest confluence criterion without a termination precondition that can be effectively used in practice. Furthermore, it also applies to terminating systems, which may be attractive if one does not know how to (effectively) show termination or if the correctness of the technique for proving termination requires confluence.

As our rewrite systems consist of constructor rules and defining rules the problem of establishing confluence of the whole rewrite system can be decomposed into three smaller sub-problems: Firstly, we show confluence of $\longrightarrow_{R^C}$, then commutation of the constructor rules with the defining rules, and finally, using these assumptions, confluence of $\longrightarrow_R$. Thus, different criteria may be applied to handle these sub-problems. For example, unless it is trivial, proving confluence of $\longrightarrow_{R^C}$ may often call for sophisticated semantic considerations or confluence criteria that apply to terminating rewrite systems only. For $\longrightarrow_R$, however, neither semantic confluence criteria nor confluence criteria with termination preconditions are practically feasible in general. One reason for this may be that effective applications of semantic confluence criteria require the specification given by the whole rewrite system to have been modeled before in some formalism. Another reason is that termination of the whole rewrite system may not be given or difficult to show without any confluence assumptions.

Since the confluence criterion we present in this section presupposes confluence of $\longrightarrow_{R^C}$, we first discuss how to establish that $\longrightarrow_{R^C}$ is confluent. Without constructor rules, i.e. $R^C = \emptyset$, confluence of $\longrightarrow_{R^C}$ is trivial. While this seems rather restrictive, this case of *free constructors* is very important in practice since a lot of data structures are freely generated. Besides, non-free constructors pose serious problems in most frameworks for inductive theorem proving — if they can be handled at all. Another way to prove confluence of $\longrightarrow_{R^C}$ is to use one of the known confluence criteria for positive conditional rewrite systems described e.g. by Dershowitz et al. (1988). Note that the application of most of these confluence criteria requires termination of $\longrightarrow_{R^C}$. Termination of the constructor rules, however, does not mean termination of the whole rewrite system. More syntactic criteria for confluence of $\longrightarrow_{R^C}$ are provided by Wirth (1995, Section 15). Sometimes, however, confluence of $\longrightarrow_{R^C}$ can only be shown by using the semantic knowledge of the specifier: either with semantic confluence criteria in the style introduced by Plaisted (1985) (see Theorem 6.5 by Wirth & Gramlich, 1994b) or in a specific way that may only work for the given set of constructor rules.

Before formally presenting our syntactic confluence criterion we have to introduce more notions concerning (conditional) critical pairs and properties of rewrite systems.

Let $t_1, t_2 \in \mathcal{T}(sig, V)$. We call a constructor substitution σ a *unifier* of $t_1 = t_2$ if $t_1\sigma = t_2\sigma$. A unifier σ of $t_1 = t_2$ is said to be *most general* on a finite set $X \subseteq V$ if for every unifier μ of $t_1 = t_2$ there is a constructor substitution τ such that $(\sigma\tau)|_X = \mu|_X$. Note that if $t_1 = t_2$ has a unifier, then it also has a most general unifier on X, which

[3]See Theorems 68 and 71 by Wirth (1995). Theorem 68 is the version for ω-shallow confluence, Theorem 71 the one for ω-level confluence.

we denote by $\mathrm{mgu}(t_1 = t_2,\ X)$.

Definition 3.3.1 Let $spec = (sig, C, R)$ be a specification with constructors and $R = R^C \uplus R^D$ where R^C is a set of constructor rules and R^D a set of defining rules.

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

$$\big(\ (l_1[r_0]_p\sigma,\ \Delta_0\sigma,\ a_0),\ (r_1\sigma,\ \Delta_1\sigma,\ a_1)\ \big)$$

is a (non-trivial) *critical pair of the form* (a_0, a_1) where, for $i \in \{0,1\}$, $a_i = 0$ if $l_i = r_i \leftarrow \Delta_i$ is in R^C and $a_i = 1$ if $l_i = r_i \leftarrow \Delta_i$ is in R^D.
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

(a) there are $u, v \in \mathcal{T}(sig, V)$ and an $i \in \{0,1\}$ such that $u = v$ or $v = u$ occurs in $\Delta_i\sigma$ and $u \neq v$ occurs in $\Delta_{1-i}\sigma$; or

(b) there are $t, \hat{u}, \hat{v} \in \mathcal{T}(sig, V)$ such that $\hat{u}$ and $\hat{v}$ are distinct $\longrightarrow_R$-irreducible ground terms, $t = \hat{u}$ or $\hat{u} = t$ occurs in $\Delta_0\sigma$ and $t = \hat{v}$ or $\hat{v} = t$ occurs in $\Delta_1\sigma$.

Consider e.g. the critical pairs of the rewrite system in Example 2.2.1: There are only two critical pairs resulting from overlapping the two div-rules into each other. Since the two critical pairs are symmetric and the notion of complementarity is symmetric too, we just have to consider one of the critical pairs, say

$$\big(\ (\mathsf{0}, y \neq \mathsf{0} \wedge \mathsf{less}(x,y) = \mathsf{true}, 1), (\mathsf{s}(\mathsf{div}(\mathsf{minus}(x,y),y)), y \neq \mathsf{0} \wedge \mathsf{less}(x,y) = \mathsf{false}, 1)\ \big).$$

It is complementary according to Definition 3.3.1(ii)(b) (instantiate t with $\mathsf{less}(x,y)$, $\hat{u}$ with true and $\hat{v}$ with false). Similarly, for the complementarity of the critical pairs of the rewrite system of Example 2.2.2 we only have to check that the critical pair

$$\big(\ (z_1, x = z_2, 1), (\mathsf{div1}(x, y, \mathsf{s}(z_1), \mathsf{plus}(z_2, y)), x \neq z_2, 1)\ \big)$$

is complementary, which is obviously the case (see Definition 3.3.1(ii)(a)).

In addition, two further notions are presupposed in our confluence criterion, namely left-linearity and a weakened form of normality. These properties of rewrite systems are necessary for establishing confluence of non-terminating rewrite systems (see above).

A rewrite system R is called *left-linear* if the left-hand side l of each rewrite rule $l = r \leftarrow \Delta$ in R is linear. Moreover, we call R *weakly normal* if each $l = r \leftarrow \Delta$ in R satisfies the following condition: For each $t_1 = t_2$ in Δ there is an $i \in \{1,2\}$ such that t_i is a $\longrightarrow_R$-irreducible ground term or $t_i \in \mathcal{T}(sig^C, V^C)$ or $\mathrm{def}(t_i)$ occurs in Δ.

Obviously, the rewrite systems in Examples 2.2.1 and 2.2.2 are left-linear. Assuming that all variables are constructor variables these rewrite systems are also weakly normal, because the right-hand sides of all equational conditions are constructor terms. Thus, we can directly apply the following syntactic confluence criterion to prove confluence of the rewrite relations in these examples.

Theorem 3.3.2 *Let $spec = (sig, C, R)$ be a specification with constructors such that R is left-linear as well as weakly normal and $R = R^C \uplus R^D$ where R^C is a set of constructor rules and R^D a set of defining rules.*
Assume that $\longrightarrow_{R^C}$ is confluent. If each critical pair in $\mathrm{CP}(R)$ of the form $(0,1)$, $(1,0)$ or $(1,1)$ is complementary, then $\longrightarrow_R$ is confluent.

Note that this confluence criterion is stronger than a similar theorem by Bergstra and Klop (1986)[4], since instead of normality it only requires weak normality, a property that is less restrictive because we can always achieve it by adding definedness atoms to the condition literals. Moreover, instead of requiring orthogonality, our theorem can deal with critical pairs provided they are complementary — a property which could again be weakened, but (to our knowledge) not in an effective manner that would satisfy the practical requirements of an admissibility condition for a specification language.

3.4 Inductive Validity

In order to define inductive validity of a clause w.r.t. a specification with constructors we have to decide what model class is to be associated with it as its inductive semantics. For an admissible specification *spec* there appear to be two appropriate choices[5], namely (i) DMod(*spec*) and (ii) (the isomorphism class of) the standard data model $\mathcal{M}(spec)$.

As inductive proofs often call for extensions of the specification in order to facilitate the formulation of missing lemmas or stronger induction hypotheses, an essential requirement for an inductive semantics is its monotonicity of validity w.r.t. "consistent" extension of the specification. For DMod(*spec*), this monotonicity property is given as is shown in the following.

Definition 3.4.1 For $i \in \{0,1\}$ let $spec_i = (sig_i, C_i, R_i)$ be a specification with constructors where $sig_i = (S_i, F_i, \alpha_i)$, and let $V_i = (V_{i,s})_{s \in S_i}$ be a variable system for sig_i. We say that $spec_1$ is a *constructor-consistent extension* of $spec_0$ 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) $R_0 \subseteq R_1$ (4) for each $c \in C_1 \backslash C_0$ with $\alpha_1(c) = s_1 \ldots s_n s$ we have $s \notin S_0$ and (5) for each constructor rule $l = r \leftarrow \Delta$ in $R_1 \backslash R_0$ we have $l \notin \mathcal{T}(sig_0^{C_0}, V_0)$.

Theorem 3.4.2 *For $i \in \{0,1\}$ let $spec_i = (sig_i, C_i, R_i)$ be an admissible specification. Assume that $spec_1$ is a constructor-consistent extension of $spec_0$, and let Γ be a clause (over sig_0 and V_0). Now if $DMod(spec_0) \models \Gamma$, then $DMod(spec_1) \models \Gamma$.*

Theorems 3.2.8 and 3.4.2 imply a similar monotonicity for the validity in $\mathcal{M}(spec)$, which, however, requires the definedness of the terms in the negative literals of the clauses. This requirement is in fact needed: The clause $\mathsf{minus}(0, \mathsf{s}(0)) \neq 0$ is valid in $\mathcal{M}(spec_{\mathsf{div}})$ (but not in DMod($spec_{\mathsf{div}}$)) where $spec_{\mathsf{div}}$ is the admissible specification of Example 2.2.1. However, after a constructor-consistent extension of $spec_{\mathsf{div}}$ with the

[4] which is also mentioned by Dershowitz et al. (1988)

[5] For a discussion of other choices see Wirth and Gramlich (1994b) or Wirth (1997).

defining rule $\mathsf{minus}(0, \mathsf{s}(y)) = 0$ the clause $\mathsf{minus}(0, \mathsf{s}(0)) \neq 0$ is no longer valid in the resulting standard data model.

Mainly because of Theorem 3.4.2 we prefer DMod(*spec*) as the inductive semantics of our specification language. Thus, we define

Definition 3.4.3 We say that a clause Γ is *inductively valid* or an *inductive theorem* with respect to an admissible specification *spec* if $\mathrm{DMod}(spec) \models \Gamma$.

Observe that the specification with constructors $spec_{\mathsf{div1}}$ from Example 2.2.2 is also admissible, which can be easily shown with Theorem 3.3.2. Now — in spite of the existence of "junk terms" — clauses such as $\mathsf{plus}(x, y) = \mathsf{plus}(y, x)$ or (2.1) are in fact inductively valid w.r.t. $spec_{\mathsf{div1}}$. Furthermore, the "invariant" for a proof of clause (2.1), namely

$$y = 0 \;\vee\; \mathsf{less}(x, \mathsf{times}(y, z)) = \mathsf{true} \;\vee\; \mathsf{div1}(x, y, 0, 0) = \mathsf{div1}(x, y, z, \mathsf{times}(y, z)) \tag{3.1}$$

is also valid in $\mathrm{DMod}(spec_{\mathsf{div1}})$ (as can be proved by induction on z) and hence an inductive theorem w.r.t. $spec_{\mathsf{div1}}$ (see Section 7.4).

Finally note that the specification language proposed in this chapter evidently meets the requirements for the specification language of our formal framework for inductive theorem proving as they were discussed in Section 2.2.1.

Chapter 4

Counterexamples, the Induction Order and Clauses with Order Literals

Before we can introduce the concrete inference rules of our calculus for inductive proofs and proof state graphs in the subsequent chapters, we have to present an underlying induction order as the second major component of the proposed formal framework for inductive theorem proving. To be more precise, in this chapter we (i) discuss fundamental concepts required for showing the *soundness* of our approach, (ii) provide a *semantic* induction order and (iii) extend clauses by a third kind of literals, namely so-called order literals. Basic requirements concerning the induction order were dealt with in Section 2.2.2. Prior to reading this chapter, the reader may refer to Section 2.3 again, which conveys an intuitive impression of how one can construct proofs of inductive theorems within our formal framework.

This chapter is organized as follows: As a conceptual basis for the soundness proof of our formalization of inductive theorem proving, the notion of an $\mathcal{A}$-*counterexample* (for a clause in a data model $\mathcal{A}$ of the specification *spec*) is introduced in 4.1. Furthermore, we give an abstract characterization of the induction order $\precsim_{\mathcal{A}}$ associated with every data model $\mathcal{A}$, specify the general form of inference rules and define the soundness of inference rules. Thereafter (in 4.2), a concrete *semantic* induction order is proposed together with a corresponding notion of a weight. In 4.3 finally, we extend clauses as defined in the preceding chapter by so-called *order literals*. This third kind of literal is a prerequisite for representing and verifying the order conditions, which "guard" applications of induction hypotheses, in our calculus for inductive proofs.

4.1 $\mathcal{A}$-Counterexamples and Inference Rules

The soundness of our formalization of inductive theorem proving will be established in Section 6.2.1 by proving that the existence of a proof graph for a goal $\langle \Gamma\,;\, w\rangle$ implies inductive validity of Γ. Recall from Section 3.4 that a clause Γ is called an inductive theorem w.r.t. an admissible specification *spec* iff $\mathcal{A} \models \Gamma$ for every data model $\mathcal{A}$ of *spec*. A sufficient (and necessary) condition for $\mathcal{A} \models \Gamma$ is that none of the so-called $\mathcal{A}$-*counterexamples* exist for Γ.

Given a data model $\mathcal{A}$ of *spec*, an $\mathcal{A}$-counterexample for a clause Γ is to describe a (ground) instance of Γ that is not valid in $\mathcal{A}$. Let σ be an inductive substitution (see Section 3.1.1). Then general variables may still occur in the instance $\Gamma\sigma$. Since there are data models that are not term-generated, such as $\mathcal{M}(\mathit{spec})$ for example, an $\mathcal{A}$-counterexample for Γ must include a "semantic" component, i.e. a valuation for the general variables in $\Gamma\sigma$, in addition to σ:

Lemma 4.1.1 *If $\mathcal{A}$ is a sig-algebra and Γ a clause then $\mathcal{A} \not\models \Gamma$ iff there is an inductive substitution σ and a valuation φ of V^{G} in $\mathcal{A}$ such that $\mathcal{A}$ does not satisfy $\Gamma\sigma$ with φ.*

We call $(\Gamma, \sigma, \varphi)$ an $\mathcal{A}$-*counterexample (for Γ)* then.

For the proof of our central soundness result Theorem 6.2.4, we will formulate a suitable *invariant* for the construction of proof state graphs (see Lemma 6.2.2) such that Theorem 6.2.4 will be an immediate consequence of this invariant. Now Lemma 6.2.2 will be proved by a well-founded *induction on $\mathcal{A}$-counterexamples* for every data model $\mathcal{A}$ of *spec*. Hence, in order to make such inductions possible, we have to provide, for every data model $\mathcal{A}$ of *spec*, a well-founded quasi-order $\precsim_{\mathcal{A}}$ on the set of $\mathcal{A}$-counterexamples, which we call the *induction order* associated with $\mathcal{A}$.

In order to facilitate the definition of an induction order $\precsim_{\mathcal{A}}$ that is as powerful and flexible as needed in practical induction proofs, we add so-called *weights* to the clauses in $\mathcal{A}$-counterexamples. A pair $\langle \Gamma ;\, w \rangle$ consisting of a clause Γ and a weight w is called a *goal*. Intuitively speaking, the weight w in a goal $\langle \Gamma ;\, w \rangle$ is a syntactic object related to Γ that can be used to *focus* on those parts of Γ that are relevant for determining appropriate "sizes" of $\mathcal{A}$-counterexamples for Γ w.r.t. $\precsim_{\mathcal{A}}$. Without concretizing the notion of a weight at this point we now define $\mathcal{A}$-counterexamples for goals in addition to $\mathcal{A}$-counterexamples for clauses.

Definition 4.1.2 Suppose $\mathcal{A}$ is a *sig*-algebra and $\langle \Gamma ;\, w \rangle$ a goal. If σ is an inductive substitution and φ a valuation of V^{G} in $\mathcal{A}$ such that $\mathcal{A}$ does not satisfy $\Gamma\sigma$ with φ, then $(\langle \Gamma ;\, w \rangle, \sigma, \varphi)$ is said to be a $\mathcal{A}$-*counterexample (for $\langle \Gamma ;\, w \rangle$)*.

By Lemma 4.1.1, $\mathcal{A} \models \Gamma$ iff there is no $\mathcal{A}$-counterexample of the form $(\langle \Gamma ;\, w \rangle, \sigma, \varphi)$. [1] We provide a concrete notion of a weight and a corresponding induction order in Section 4.2.

[1] Note that in the (proofs of the) soundness results in Section 6.2.1 no use will be made of any concrete property of the weights in $\mathcal{A}$-counterexamples. All that will be used there is the preceding remark, the existence of a *well-founded quasi-order* $\precsim_{\mathcal{A}}$ on the set of $\mathcal{A}$-counterexamples for every data model $\mathcal{A}$ of *spec*, and an abstract characterization of *sound* inference rules (see Definition 4.1.4).

Let us now describe those abstract properties of the concrete *inference rules* to be presented in Chapter 5 that will be used in the proof of the invariant for the construction of proof state graphs (see Lemma 6.2.2). Every inference rule of our calculus for inductive proofs can be characterized by the following general scheme:

Notation 4.1.3 An inference rule has the form

<rule name>

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

if <applicability-conditions>

where $n, k \in \mathbb{N}$ and $U_j \in \{\mathcal{I}, \mathcal{L}\}$ for $j = 1, \ldots, k$. This notation is to express that, with the inference rule named <rule name>, a goal $\langle \Gamma\,;\,w\rangle$ can be reduced to a sequence of new (sub-) goals $\langle \Gamma_1\,;\,w_1\rangle, \ldots, \langle \Gamma_n\,;\,w_n\rangle$ provided that the <applicability-conditions> of this inference rule are satisfied. For the reduction of $\langle \Gamma\,;\,w\rangle$ with <rule name>, the inference rule may make use of further goals $\langle \Pi_1\,;\,\hat{w}_1\rangle, \ldots, \langle \Pi_k\,;\,\hat{w}_k\rangle$ $(k \geq 0)$ by applying these goals to $\langle \Gamma\,;\,w\rangle$ in one of two possible ways: For each of the goals $\langle \Pi_j\,;\,\hat{w}_j\rangle$, the annotation U_j indicates whether $\langle \Pi_j\,;\,\hat{w}_j\rangle$ is applied as an *induction hypothesis* $(U_j = \mathcal{I})$ or *non-inductively* $(U_j = \mathcal{L})$. In the latter case, Π_j is (the clause representation of) a defining rule or a (possibly unproved) lemma. If $k > 0$ then the inference rule is called *applicative*, otherwise *non-applicative*.

We ask the reader to refer to Chapter 5 for examples of concrete inference rules.

The soundness property of an inference rule as required in the proof of the invariant (Lemma 6.2.2) is captured in the following definition. It describes the effect of a sound inference rule abstractly in terms of $\mathcal{A}$-counterexamples and the induction order $\precsim_{\mathcal{A}}$. Basically it states that applying a sound inference rule to a goal $\langle \Gamma\,;\,w\rangle$ for which an $\mathcal{A}$-counterexample $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$ exists gives rise to another $\mathcal{A}$-counterexample for one of the subgoals or goals applied as induction hypotheses — given that the non-inductively applied goals actually contain inductively valid clauses. Moreover, the new $\mathcal{A}$-counterexample must be smaller than or equivalent to $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$ then.

Definition 4.1.4 We call an inference rule *sound* if for any admissible specification *spec*, 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 $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$, 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 $(\langle \Gamma_i\,;\,w_i\rangle, \tau, \psi) \precsim_{\mathcal{A}} (\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$.

(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 $(\langle \Pi_j \,;\, \hat{w}_j \rangle, \tau, \psi) \prec_{\mathcal{A}} (\langle \Gamma \,;\, w \rangle, \sigma, \varphi)$.

We will have to show that each of the inference rules introduced in Chapter 5 fulfills this soundness condition. In these "local" soundness proofs we will use specific properties of the weights and the induction order $\precsim_{\mathcal{A}}$ as defined in the following section.

4.2 Weights and the Semantic Induction Order $\precsim_{\mathcal{A}}$

After the *abstract* characterization of $\mathcal{A}$-counterexamples, the induction order and sound inference rules, we now propose a *concrete* semantic induction order $\precsim_{\mathcal{A}}$ for every data model $\mathcal{A}$ of an admissible specification together with a corresponding notion of a weight. The essential technical idea underlying weights and our semantic induction order $\precsim_{\mathcal{A}}$ is quite simple and can be stated as follows:[2]

A weight w in a $\mathcal{A}$-counterexample $(\langle \Gamma \,;\, w \rangle, \sigma, \varphi)$ will be a *syntactic* object, namely a k-tuple of terms, that evaluates to a *semantic* object $\mathrm{eval}_{\varphi}^{\mathcal{A}}(w\sigma)$, i.e. a k-tuple of elements of the carriers of $\mathcal{A}$. This semantic object can be considered the *size* of $(\langle \Gamma \,;\, w \rangle, \sigma, \varphi)$. By providing an order on the semantic objects used as the sizes of $\mathcal{A}$-counterexamples we obtain our *semantic* induction order $\precsim_{\mathcal{A}}$. This order will be the lexicographical order $\leq_{\mathcal{A}}^{\mathrm{lex}}$ that is induced by a *well-founded partial order* $\leq_{\mathcal{A}}$ defined on the carriers of every data model $\mathcal{A}$ of an admissible specification with *free* constructors.

Since the lexicographical order induced by a well-founded partial order is not well-founded in general (see Section 2.4), we have to restrict the length of the k-tuples to be used as weights by a sufficiently big constant k_0. Indeed, $\leq_{\mathcal{A}}^{\mathrm{lex}}$ is a well-founded partial order on the set of k_0-bounded k-tuples of elements of $\mathcal{A}$. So in the remainder of the paper, let k_0 be a *big* fixed natural number.

Definition 4.2.1 Let $sig = (S, F, \alpha)$ be a signature such that $C \subseteq F$ is a set of constructors for sig. A *weight* (over sig and V) is a sequence of terms $(t_1, \ldots, t_k)$ such that $0 \leq k \leq k_0$ and $t_1, \ldots, t_k \in \mathcal{T}(sig, V)$. Instead of (t) we usually write t. The set of weights (over sig and V) is denoted by $\mathcal{W}(sig, V)$.

As mentioned above, a weight $(t_1, \ldots, t_k)$ can be evaluated to a k-tuple of elements of $\mathcal{A}$ by means of the usual evaluation mechanism. Let φ be a valuation of V in $\mathcal{A}$. For $w \in \mathcal{W}(sig, V)$ with $w = (t_1, \ldots, t_k)$, we define $\mathrm{eval}_{\varphi}^{\mathcal{A}}(w) = (\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1), \ldots, \mathrm{eval}_{\varphi}^{\mathcal{A}}(t_k))$.

So given a conjecture in form of a clause Γ, what is a suitable weight w for Γ in the goal $\langle \Gamma \,;\, w \rangle$? While there does not seem to be a universally applicable and effective method of determining suitable weights for conjectures, we can derive an approximate idea

[2] The somewhat technical presentation of weights and the semantic induction order in this section is supplemented by a motivating discussion at the end of Section 4.3.

from the characterization of sound inference rules (see Definition 4.1.4): Essentially, the choice of w for Γ must make sure that order conditions (expressed in terms of $\precsim_{\mathcal{A}}$) arising from applications of induction hypotheses in the proof of $\langle \Gamma\,;\, w\rangle$ are *provably* fulfilled. That is, if there is an $\mathcal{A}$-counterexample for a goal in the proof of $\langle \Gamma\,;\, w\rangle$ to which a goal is applied as induction hypothesis, then there must be a corresponding $\mathcal{A}$-counterexample for the applied induction hypothesis that can be shown to be smaller (w.r.t. $\prec_{\mathcal{A}}$). In numerous practically relevant cases, these "termination" proofs succeed if a tuple of terms is used as weight for Γ that comprises the so-called *induction variables* of Γ. In other cases, a weight w may only be suitable for Γ if w represents applications of appropriate *measure functions* to the induction variables of Γ. Instead of formally defining notions of induction variables and measure functions (see Walther, 1994; Boyer & Moore, 1979), we appeal to the reader's intuition with the following examples.

Example 4.2.2 Consider the admissible specification $spec_{\mathsf{div}}$ from Example 2.2.1. Suppose we are to prove the inductive validity of $\mathsf{plus}(\mathsf{plus}(x,y),z) = \mathsf{plus}(x,\mathsf{plus}(y,z))$ w.r.t. $spec_{\mathsf{div}}$ where $x,y,z \in V^{\mathrm{C}}$. It is commonly known that this conjecture should be proved by (what is intuitively called) *induction on z* — mainly because plus is defined by recursion on the second argument. So by using the induction variable z as weight, we can easily show that the order condition resulting from the application of the conjecture

$$\langle\, \mathsf{plus}(\mathsf{plus}(x,y),z) = \mathsf{plus}(x,\mathsf{plus}(y,z)) \;;\; z\rangle \tag{4.1}$$

as induction hypothesis to the goal

$$\langle \ldots \vee \mathsf{s}(\mathsf{plus}(\mathsf{plus}(x,y),z)) = \mathsf{s}(\mathsf{plus}(x,\mathsf{plus}(y,z))) \;;\; \mathsf{s}(z)\rangle \tag{4.2}$$

in the induction step is actually fulfilled (see the discussion in Section 4.3).

Example 4.2.3 To exemplify the need of k-tuples as weights ($k > 1$), let us extend the specification $spec_{\mathsf{div}}$ from Example 2.2.1 by axioms for *Ackermann's function*:

$$\begin{array}{lcl}
\mathsf{ack}(0,y) & = & \mathsf{s}(y)\\
\mathsf{ack}(\mathsf{s}(x),0) & = & \mathsf{ack}(x,\mathsf{s}(0))\\
\mathsf{ack}(\mathsf{s}(x),\mathsf{s}(y)) & = & \mathsf{ack}(x,\mathsf{ack}(\mathsf{s}(x),y))
\end{array}$$

Even simple inductive theorems such as $\mathrm{def}(\mathsf{ack}(x,y))$ (i.e. the total definedness of ack) or $\mathsf{less}(y,\mathsf{ack}(x,y)) = \mathsf{true}$ call for proofs by *(nested) induction on x and y*. Using the tuple (x,y) as weight we can construct proofs for the goals $\langle \mathrm{def}(\mathsf{ack}(x,y))\,;\,(x,y)\rangle$ and $\langle \mathsf{less}(y,\mathsf{ack}(x,y)) = \mathsf{true}\,;\,(x,y)\rangle$ in our framework.

Example 4.2.4 To illustrate the use of a measure function we extend the specification from Example 2.2.1 by a new sort list for lists of natural numbers with constructors nil and cons and by the following defining rules for the operations app and len:

$$\begin{array}{lcl}
\mathsf{app}(\mathsf{nil},l) & = & l\\
\mathsf{app}(\mathsf{cons}(x,l_1),l_2) & = & \mathsf{cons}(x,\mathsf{app}(l_1,l_2))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{len}(\mathsf{nil}) & = & 0\\
\mathsf{len}(\mathsf{cons}(x,l)) & = & \mathsf{s}(\mathsf{len}(l))
\end{array}$$

Consider the inductive theorem $\mathsf{app}(l, \mathsf{nil}) = l$. We can prove it either by (structural) induction on l or by induction on the *length of the list represented by l*. In the former case, the goal to prove is $\langle\, \mathsf{app}(l, \mathsf{nil}) = l \;;\; l \rangle$; in the latter case, we apply the *measure function* len to l to obtain the goal $\langle\, \mathsf{app}(l, \mathsf{nil}) = l \;;\; \mathsf{len}(l) \rangle$.[3]

Since the subject of Part I is the formal framework for inductive theorem proving underlying QUODLIBET, a more systematic — yet still heuristic — approach to the determination of suitable weights (without measure functions) for conjectures will be dealt with in Part II (see Section 8.3.4).

As was mentioned in the beginning of this section, the lexicographical order $\leq_{\mathcal{A}}^{\mathrm{lex}}$ induced by a well-founded partial order $\leq_{\mathcal{A}}$ on the carriers of a data model $\mathcal{A}$ will be used to compare the semantic objects $\mathrm{eval}_{\varphi}^{\mathcal{A}}(w\sigma)$ and $\mathrm{eval}_{\varphi'}^{\mathcal{A}}(w'\sigma')$ corresponding to the $\mathcal{A}$-counterexamples $(\langle \Gamma \;;\; w \rangle, \sigma, \varphi)$ and $(\langle \Gamma' \;;\; w' \rangle, \sigma', \varphi')$. We now define the relation $\leq_{\mathcal{A}}$ associated with a *sig*-algebra $\mathcal{A}$ and show that $\leq_{\mathcal{A}}$ is a well-founded partial order on the carriers of $\mathcal{A}$ if $\mathcal{A}$ is a data model of an admissible specification with *free* constructors.

Definition 4.2.5 Let $sig = (S, F, \alpha)$ be a signature such that $C \subseteq F$ is a set of constructors for *sig*, and let $\mathcal{A} = (A, F^{\mathcal{A}})$ be a *sig*-algebra. The relation $\leq_{\mathcal{A}}$ *associated with* $\mathcal{A}$ is defined on A by $a_1 \leq_{\mathcal{A}} a_2$ if

(a) $a_1 = a_2$; or

(b) there are $t_1, t_2 \in \mathcal{GT}(sig^C)$ such that $t_i^{\mathcal{A}} = a_i$ for $i = 1, 2$ and $|t_1| < |t_2|$.

If $a_1 <_{\mathcal{A}} a_2$ then both a_1 and a_2 are elements of the data reduct $\mathcal{A}^C$ of $\mathcal{A}$. Note that $\leq_{\mathcal{A}}$ is only a well-founded partial order on A if every element of $\mathcal{A}^C$ is *uniquely* represented by a constructor ground term.[4] Observe also that $\leq_{\mathcal{A}}$ is not sort-invariant in general.

An admissible specification $spec = (sig, C, R)$ is an admissible specification with *free constructors* if R is a set of defining rules, that is if $R^C = \emptyset$. By Proposition 3.2.5(2), freeness of constructors is sufficient for the unique representation of the elements of $\mathcal{A}^C$ by constructor ground terms.

Lemma 4.2.6 *Let spec be an admissible specification with free constructors, and let $\mathcal{A} = (A, F^{\mathcal{A}})$ be a sig-algebra such that $\mathcal{A} \in DMod(spec)$. Then $\leq_{\mathcal{A}}$ is a well-founded partial order on A.*

Note that any well-founded partial order on $\mathcal{GT}(sig^C)$ — such as e.g. the syntactic reduction order RPO (see Dershowitz, 1987) — could be used for the comparison of the constructor ground terms t_1 and t_2 in Definition 4.2.5 (b). Thus far, however,

[3] Observe that for a proof of the goal $\langle\, \mathrm{def}(\mathsf{flatten}(t)) \;;\; w \rangle$, which states the total definedness of a $\mathsf{flatten}$ operation on binary trees, a weight w involving a measure function is actually necessary (see Section E.3.3 for the definition of $\mathsf{flatten}$, the used measure function and a proof of the goal).

[4] For otherwise, $\leq_{\mathcal{A}}$ may be a trivial relation: Let $spec = (sig, C, R)$ be the admissible specification defined by $S = \{\mathsf{int}\}$, $F = C = \{\mathsf{0}, \mathsf{s}, \mathsf{p}\}$, and $R = R^C = \{\, \mathsf{s}(\mathsf{p}(z)) = z \,,\, \mathsf{p}(\mathsf{s}(z)) = z \,\}$. Then $[t_1] <_{\mathcal{M}(spec)} [t_2]$ holds for *any* $t_1, t_2 \in \mathcal{GT}(sig^C)$ where $\mathcal{M}(spec)$ is the standard data model of *spec*.

there has been no need for a more sophisticated term order than the one induced by the length of terms; despite its simplicity, the latter yields a well-founded order $\leq_{\mathcal{A}}$ which is sufficiently powerful for most practically relevant examples (as will be demonstrated in the remainder of the thesis).

By making use of the lexicographical order $\leq_{\mathcal{A}}^{\mathrm{lex}}$ induced by $\leq_{\mathcal{A}}$ we can finally define our semantic induction order $\precsim_{\mathcal{A}}$. Its well-foundedness essentially follows from Lemma 4.2.6 and the fact that the length of the k-tuples used as weights is bounded by k_0.

Definition 4.2.7 Let $\mathcal{A}$ be a data model of an admissible specification *spec* with free constructors. We define the *induction order* $\precsim_{\mathcal{A}}$ associated with $\mathcal{A}$ on the set of $\mathcal{A}$-counterexamples by

$$(\langle \Gamma \,;\, w\rangle, \sigma, \varphi) \precsim_{\mathcal{A}} (\langle \Gamma' \,;\, w'\rangle, \sigma', \varphi') \quad \text{if} \quad \mathrm{eval}_{\varphi}^{\mathcal{A}}(w\sigma) \leq_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_{\varphi'}^{\mathcal{A}}(w'\sigma')$$

Theorem 4.2.8 *Let $\mathcal{A}$ be a data model of an admissible specification spec with free constructors. Then $\precsim_{\mathcal{A}}$ is a well-founded quasi-order on the set of $\mathcal{A}$-counterexamples.*

It should be observed that the restriction to free constructors which we have to impose from now on is rather common in the field of first-order inductive theorem proving (see Boyer & Moore, 1979; Walther, 1994; Bouhoula & Rusinowitch, 1995). For a discussion of the severe problems caused by an attempt to formalize and validate the integers using the non-free constructors $\mathsf{0}$, s and p we refer to (Wirth & Kühler, 1995, pp. 73–74).

4.3 Clauses with Order Literals

Before we conclude this chapter by providing evidence for the practical usefulness of our semantic induction order, we first introduce a third kind of literals in clauses, namely so-called order literals. An *order atom* is an expression of the form $w_1 < w_2$, where w_1 and w_2 are weights, and can be used to represent the order condition arising from an application of a goal as induction hypothesis to another goal (see Definition 4.1.4). This extension of the set of clauses that are to be admitted by our calculus for inductive proofs is a prerequisite for integrating the induction orders associated with the data models of a specification into our calculus. In the subsequent example we will try to convey how advantageous this integration may be, for then the complete inductive reasoning mechanism will be available for the proofs of these order conditions.

Example 4.3.1 Let *spec* be the admissible specification from Example 2.2.1 (note that its constructors are free). Consider the "domain lemma" for the division operation, that is the clause in the goal

$$\langle\, \mathrm{def}(\mathsf{div}(x,y)) \,\vee\, y = \mathsf{0} \,;\, x\rangle \tag{4.3}$$

As div is defined by *destructor*[5] recursion, this lemma can be easily proved by destructor induction on x. In the induction step of the proof, the instance of the original goal (4.3) given by the matching substitution $\{x \leftarrow \mathsf{minus}(x,y)\}$

$$\langle\, \mathrm{def}(\mathsf{div}(\mathsf{minus}(x,y),y)) \,\vee\, y=\mathsf{0} \;;\; \mathsf{minus}(x,y)\rangle$$

should be (subsumptively) applied as induction hypothesis to the goal

$$\langle\, \mathrm{def}(\mathsf{div}(\mathsf{minus}(x,y),y)) \vee \ldots \vee \mathsf{less}(x,y)=\mathsf{true} \vee \ldots \vee y=\mathsf{0} \;;\; x\rangle \tag{4.4}$$

The order condition "guarding" this application of an induction hypothesis is provably fulfilled if, for any data model $\mathcal{A}$ of *spec* and for any $i,j\in\mathbb{N}$, the following can be shown (see Definition 4.1.4):

$$\mathsf{minus}(\mathsf{s}^i(\mathsf{0}),\mathsf{s}^j(\mathsf{0}))^{\mathcal{A}} <_{\mathcal{A}} \mathsf{s}^i(\mathsf{0})^{\mathcal{A}} \quad \text{if} \quad \mathsf{less}(\mathsf{s}^i(\mathsf{0}),\mathsf{s}^j(\mathsf{0}))^{\mathcal{A}} \neq \mathsf{true}^{\mathcal{A}} \quad \text{and} \quad \mathsf{s}^j(\mathsf{0})^{\mathcal{A}} \neq \mathsf{0}^{\mathcal{A}} \tag{4.5}$$

Observe that (4.5) is a *conditional* "termination" statement. Furthermore, verifying this order condition evidently requires an inductive proof. Hence, it appears both useful and necessary to integrate the kind of reasoning needed for the verification of such order conditions into the calculus for inductive proofs.

As a first step to achieve this integration, we will provide the syntax and semantics of clauses with order literals below. Note that the formal counterpart of (4.5), namely the clause $\mathsf{minus}(x,y)<x \,\vee\, \mathsf{less}(x,y)=\mathsf{true} \,\vee\, y=\mathsf{0}$ will be an inductive theorem w.r.t. *spec*. In Chapter 5 we will complete the integration by presenting inference rules with the following properties (among others): Firstly, each of the two inference rules allowing applications of goals as induction hypotheses will be of the form

$$\frac{\langle \Gamma \;;\; w\rangle}{\langle \Gamma_1 \;;\; w\rangle \;\ldots\; \langle \Gamma_n \;;\; w\rangle \;\langle\, \hat{w}\mu < w \,,\, \Gamma_{n+1} \;;\; w\rangle} \quad \text{with} \quad \langle \Pi \;;\; \hat{w}\rangle^{\mathcal{I}}$$

where μ is the matching substitution applied to $\langle \Pi \;;\; \hat{w}\rangle$. The goal $\langle\, \hat{w}\mu < w \,,\, \Gamma_{n+1} \;;\; w\rangle$ will be called an *order subgoal*. Its purpose will be to represent the order condition corresponding to the application of the induction hypothesis $\langle \Pi \;;\; \hat{w}\rangle$ to the goal $\langle \Gamma \;;\; w\rangle$ and introduce it as a proof obligation. Secondly, there will also be inference rules allowing syntactic proofs of goals containing order literals.

For example, an application of (4.3) as induction hypothesis to (4.4) using the match $\mu = \{x \leftarrow \mathsf{minus}(x,y)\}$ will yield the order subgoal[6]

$$\langle\, \mathsf{minus}(x,y) < x \vee \ldots \vee \mathsf{less}(x,y)=\mathsf{true} \vee \ldots \vee y=\mathsf{0} \;;\; x\rangle \tag{4.6}$$

In Section 7.3.4, we will show how to construct a proof of (4.3) with QUODLIBET, which will include a proof of (4.6).

[5]According to the terminology used by Boyer and Moore (1979), minus is a so-called *destructor* operation.

[6]see the inference rule Inductive Subsumption in Section 5.3.2

Let us now explain the syntax and semantics of clauses as they are admitted by our calculus for inductive proofs. Equations and definedness atoms were defined in Section 3.1.1. An *order atom* is an expression of the form $w_1 < w_2$ where '$<$' is another predefined binary predicate symbol and $w_1, w_2 \in \mathcal{W}(sig, V)$. An *atom* is an equation, a definedness atom or an order atom. Given these three kinds of atoms, the definitions of literals, of the complement of literals and of clauses read exactly as the approximate definitions in Section 3.1.1. In order to define the usual semantic notions for clauses with order literals we only have to add the following sentence to case (ii) of Definition 3.1.4: A sig-algebra $\mathcal{A}$ satisfies an order atom $w_1 < w_2$ with a valuation φ of V in $\mathcal{A}$ if $\mathrm{eval}_\varphi^\mathcal{A}(w_1) <_\mathcal{A}^{\mathrm{lex}} \mathrm{eval}_\varphi^\mathcal{A}(w_2)$.

We conclude this section by providing evidence for the practical usefulness of our semantic induction order $\precsim_\mathcal{A}$, which is semantic in the sense that it is based on the well-founded partial order $\leq_\mathcal{A}$ defined on the carriers of a data model $\mathcal{A}$. Having introduced clauses with order literals before, we can now characterize relevant properties of $\precsim_\mathcal{A}$ (or rather of $<_\mathcal{A}^{\mathrm{lex}}$) in terms of inductively valid clauses with order literals. For that purpose, let $spec_{\mathsf{natlist}}$ be the admissible specification with free constructors that is defined as the result of a (constructor-consistent) extension of the specification $spec_{\mathsf{div}}$ from Example 2.2.1 by the sort list and the axioms for the operations plus, ack, app and len as they were given in Examples 4.2.2, 4.2.3 and 4.2.4.

First of all, it should be noted that order subgoals generated in proofs by *structural induction* on one constructor variable can mostly be proved very easily, since the order atoms they contain are usually simple inductive theorems. For instance, consider the induction step in a proof of the associativity of plus (see Example 4.2.2). The order subgoal $\langle\, z < \mathsf{s}(z) \ \vee \ldots ;\ \mathsf{s}(z)\rangle$ guarding the application of the induction hypothesis (4.1) to the goal (4.2) contains the order atom $z < \mathsf{s}(z)$, which is obviously inductively valid w.r.t. $spec_{\mathsf{natlist}}$, as $\mathsf{s}(z)$ is a longer constructor term than z. Structural inductions on lists do not pose any problems with regard to the induction order either. In the induction step of a proof of $\langle\, \mathsf{app}(l, \mathsf{nil}) = l \ ;\ l\rangle$, for example, the conjecture is applied as induction hypothesis to the goal $\langle\, \mathsf{cons}(x, \mathsf{app}(l, \mathsf{nil})) = \mathsf{cons}(x, l) \ ;\ \mathsf{cons}(x, l)\rangle$ giving rise to the order subgoal $\langle\, l < \mathsf{cons}(x, l) \ \vee \ldots ;\ \mathsf{cons}(x, l)\rangle$. The order atom $l < \mathsf{cons}(x, l)$ occurring in this order subgoal is clearly an inductive theorem w.r.t. $spec_{\mathsf{natlist}}$.

Secondly, we would like to point out the suitability of our semantic induction order for proofs by *(nested) induction on several variables*. Let X, Y, Z_1 and Z_2 be *general* variables, and let $\mathcal{A} = (A, F^\mathcal{A})$ be a data model of $spec_{\mathsf{natlist}}$. Since $\leq_\mathcal{A}$ is not only defined on A^C but on A and since $\leq_\mathcal{A}$ also includes pairs of the form (a, a) where $a \in A \backslash A^C$, clauses such as

$$(x, X) < (\mathsf{s}(x), Y)$$

$$(X, Z_1) < (Y, Z_2) \ \vee \ X \not< Y$$

$$(X, Z_1) < (Y, Z_2) \ \vee \ X \neq Y \ \vee \ Z_1 \not< Z_2$$

are inductive theorems w.r.t. $spec_{\mathsf{natlist}}$, which can be shown easily. In particular, the order subgoal $\langle\, (x, \mathsf{ack}(\mathsf{s}(x), y)) < (\mathsf{s}(x), \mathsf{s}(y)) \vee \ldots ;\ (\mathsf{s}(x), \mathsf{s}(y))\rangle$ generated in the proof of the goal $\langle\, \mathrm{def}(\mathsf{ack}(x,y))\ ;\ (x,y)\rangle$ (see Example 4.2.3) contains an inductively valid order atom.

Thirdly, let us emphasize that, due to our integration of the semantic induction order into the calculus for inductive proofs, our framework for inductive theorem proving supports the construction of proofs by *destructor induction* as well. Such proofs are usually required for inductive theorems which express valid properties of operations defined by destructor recursion. In Example 4.3.1, we have already discussed aspects of a proof by destructor induction as far as the induction order and order subgoals are concerned. We provide another simple, yet characteristic example:

Example 4.3.2 We extend the specification $spec_{\mathsf{natlist}}$ (see above) by a further axiomatization of the append operation on lists, this time using a definition by destructor recursion in the style of Walther (1994):

$$\begin{array}{lll} \mathsf{app1}(l_1,l_2) = l_2 & \leftarrow & l_1 = \mathsf{nil} \\ \mathsf{app1}(l_1,l_2) = \mathsf{cons}(\mathsf{car}(l_1),\mathsf{app1}(\mathsf{cdr}(l_1),l_2)) & \leftarrow & l_1 \neq \mathsf{nil} \end{array}$$

The two destructors car and cdr are (incompletely) specified by $\mathsf{car}(\mathsf{cons}(x,l))=x$ and $\mathsf{cdr}(\mathsf{cons}(x,l))=l$, respectively. Observe that this extension of $spec_{\mathsf{natlist}}$ yields an admissible specification. Now consider the goal $\langle\, \mathsf{app1}(l,\mathsf{nil})=l\ ;\ l\rangle$. We can construct a proof by destructor induction on l for this goal that contains an application of the conjecture as induction hypothesis to the goal

$$\langle \ldots \vee\ l=\mathsf{nil}\ \vee \ldots \vee\ \mathsf{cons}(\mathsf{car}(l),\mathsf{app1}(\mathsf{cdr}(l),\mathsf{nil}))=l\ ;\ l\rangle$$

using the match $\mu = \{l \leftarrow \mathsf{cdr}(l)\}$. The order subgoal resulting from this application of an induction hypothesis is of the form $\langle\, \mathsf{cdr}(l) < l\ \vee \ldots \vee\ l=\mathsf{nil}\ \vee \ldots\ ;\ l\rangle$. It is easily seen that the clause $\mathsf{cdr}(l) < l\ \vee\ l=\mathsf{nil}$ is an inductive theorem, which can be easily proved with QuodLibet.

Examples 4.3.1 and 4.3.2 suggest that for every destructor d, a suitable inductive theorem (with order atoms) should be determined that can be used to subsume the clauses in order subgoals generated in proofs by destructor induction with respect to d. Examples of such inductive theorems are the clause $\mathsf{cdr}(l) < l \vee l=\mathsf{nil}$ for the destructor cdr or the clause $\mathsf{minus}(x,y) < x\ \vee\ \mathsf{less}(x,y)=\mathsf{true}\ \vee\ y=\mathsf{0}$ for the destructor minus. Note that these inductive theorems correspond to the so-called *induction lemmas* of Boyer and Moore (1979).

Finally observe that the semantic induction order introduced in this chapter obviously meets the requirements with respect to the induction order of our formal framework for inductive theorem proving as discussed in Section 2.2.2: Our induction order does not presuppose termination of the rewrite relation $\longrightarrow_R$ associated with an admissible specification (see Theorem 4.2.8), and it is well suited for proofs by destructor induction.

Chapter 5

Inference Rules

In Section 4.3 of the foregoing chapter, we presented the syntax and semantics of clauses as admitted by our calculus for inductive proofs. Besides equational and definedness literals, we provided a third kind of literals that may occur in clauses, namely order literals. Order literals are a prerequisite for integrating the semantic induction order (defined in Section 4.2) into our calculus. In this chapter, we introduce the concrete inference rules of our calculus for inductive proofs, which constitute the setting for the *syntactic* reasoning used in proofs of inductively valid clauses (with order literals) in the proposed formal framework for inductive theorem proving. For a general discussion of the requirements that have guided the design of these inference rules we refer to Section 2.2.3.

As was already specified in Notation 4.1.3, every inference rule of our calculus for inductive proofs is of the form

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

where $n, k \in \mathbb{N}$ and $U_j \in \{\mathcal{I}, \mathcal{L}\}$ for $j = 1, \ldots, k$. That is, an inference rule can be used to reduce a goal $\langle \Gamma\,;\,w\rangle$ to new (sub-) goals $\langle \Gamma_1\,;\,w_1\rangle, \ldots, \langle \Gamma_n\,;\,w_n\rangle$. The inference rule may make use of further goals $\langle \Pi_1\,;\,\hat{w}_1\rangle, \ldots, \langle \Pi_k\,;\,\hat{w}_k\rangle$ each of which can be applied to $\langle \Gamma\,;\,w\rangle$ either as an induction hypothesis ($U_j = \mathcal{I}$) or non-inductively as an axiom or lemma ($U_j = \mathcal{L}$). Recall that an inference rule is said to be *applicative* if $k > 0$, and *non-applicative* if $k = 0$.

Since the subject of Part I of this thesis is the *formal framework* for inductive theorem proving underlying QuodLibet, a systematic (heuristic) approach to the problem of determining appropriate instances of our inference rules for the construction of proof (state) graphs for given inductive theorems is beyond the scope of this chapter. It should be noted, however, that for a significant number of the inference rules it is fairly easy to decide whether — and if so, how — to apply the inference rule to a given goal (see below).

This chapter is organized as follows: After compiling further technical notions required in the remainder of this chapter (5.1), we first introduce the non-applicative inference rules in 5.2, while the applicative inference rules are dealt with in 5.3. In Section 5.4, we assess the proposed calculus for inductive proofs with regard to the requirements discussed in Section 2.2.3. All in all, we introduce twenty-five inference rules, each of which is sound as well as *safe* (see below). Unless stated otherwise every example in this chapter is based on the admissible specification $spec_{\mathsf{natlist}}$ from Section 4.3.

5.1 Preliminaries

The "local" property of an inference rule required for the soundness proof of our formal framework in Section 6.2.1 was captured in the definition of a *sound* inference rule (see Definition 4.1.4). In order to prove the *refutational* soundness[1] of our formal framework, a further "local" property is needed, namely that of a *safe* inference rule. Roughly speaking, applying only safe inference rules in the construction of a proof state graph for a goal with an inductively valid clause cannot lead to the deduction of a goal with a contradictory clause (i.e. the empty clause).

Definition 5.1.1 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*.

A simple and often useful way to achieve safeness of an inference rule is suggested in the next lemma: An inference rule is safe if every subgoal resulting from an application of the inference rule is formed by adding literals to the clause of the goal.

Lemma 5.1.2 *If every instance of an inference rule is of the form*

$$\frac{\langle \Gamma\,;\, w\rangle}{\langle \Lambda_1, \Gamma\,;\, w_1\rangle \;\ldots\; \langle \Lambda_n, \Gamma\,;\, w_n\rangle} \quad \textit{with} \quad \langle \Pi_1\,;\, \hat{w}_1\rangle^{U_1}, \ldots, \langle \Pi_k\,;\, \hat{w}_k\rangle^{U_k}$$

where $n, k \in \mathbb{N}$, $\Lambda_1, \ldots, \Lambda_n$ *are clauses and* $U_j \in \{\mathcal{I}, \mathcal{L}\}$ *for* $j = 1, \ldots, k$, *then the inference rule is safe.*

Note that the safeness of a significant number of the inference rules proposed in this chapter follows from Lemma 5.1.2.

[1] as explained in Section 2.2.3; see Theorem 6.2.5 in Section 6.2.2

The following technical notions and notations are needed in the subsequent subsections for the presentation of the inference rules.

We first define positions of terms in literals. For an equation or definedness atom, the set of its term positions is defined as $\mathrm{Pos}(\,t_1 = t_2\,) = \{\,ip \mid i \in \{1,2\} \wedge p \in \mathrm{Pos}(t_i)\}$ and $\mathrm{Pos}(\,\mathrm{def}(t)\,) = \{\,1p \mid p \in \mathrm{Pos}(t_1)\}$; moreover, $(\,t_1 = t_2\,)/ip = t_i/p$ and $\mathrm{def}(t)/1p = t/p$. Term positions in an order atom are defined as $\mathrm{Pos}(\,(t_{1,1},\ldots,t_{1,n_1}) < (t_{2,1},\ldots,t_{2,n_2})\,) = \{\,ijp \mid i \in \{1,2\} \wedge j \in \{1,\ldots,n_i\} \wedge p \in \mathrm{Pos}(t_{i,j})\}$ and we access a term in an order atom by $(\,(t_{1,1},\ldots,t_{1,n_1}) < (t_{2,1},\ldots,t_{2,n_2})\,)/ijp = t_{i,j}/p$. For example, we have $\mathrm{Pos}(\,() < \mathsf{s}(0)\,) = \{2.1, 2.1.1\}$ (recall that '$\mathsf{s}(0)$' stands for the weight '$(\mathsf{s}(0))$'). For a negative literal λ, we define $\mathrm{Pos}(\lambda) = \mathrm{Pos}(\overline{\lambda})$ and $\lambda/p = \overline{\lambda}/p$. By $\lambda[t]_p$, we denote the literal obtained by replacing the term λ/p with a term t where $p \in \mathrm{Pos}(\lambda)$.

For some inference rules, an extended notion of equality of (equational) literals is needed: We say that $\lambda_1 =_{\mathrm{lit}} \lambda_2$ if (1) $\lambda_1 = \lambda_2$ or (2) there are $t_1, t_2 \in \mathcal{T}(sig, V)$ such that $\lambda_1 = (\,t_1 = t_2\,)$ and $\lambda_2 = (\,t_2 = t_1\,)$, or $\lambda_1 = (\,t_1 \neq t_2\,)$ and $\lambda_2 = (\,t_2 \neq t_1\,)$. By $t_1 \doteq t_2$, we mean either $t_1 = t_2$ or $t_2 = t_1$, and $t_1 \dot{\neq} t_2$ denotes either $t_1 \neq t_2$ or $t_2 \neq t_1$. We say that a literal λ *occurs* in a clause Γ, if there is a literal λ' in Γ such that $\lambda' =_{\mathrm{lit}} \lambda$, and Γ *contains* a clause Δ, if each literal in Δ occurs in Γ.

In the remainder of this chapter, we use the following (meta-) variables without further explanation: (i) m, n, k for natural numbers (ii) t, u, v, l, r for terms (iii) w for weights (iv) λ for literals and (v) $\Gamma, \Delta, \Pi, \Lambda, \Sigma$ for clauses, i.e. (possibly empty) sequences of literals.

5.2 Non-Applicative Inference Rules

As was introduced before, a non-applicative inference rule reduces a goal $\langle \Gamma\,;\, w\rangle$ to a sequence of new (sub-) goals $\langle \Gamma_1\,;\, w_1\rangle \;\ldots\; \langle \Gamma_n\,;\, w_n\rangle$ *without* making use of any further goals as induction hypotheses, axioms or lemmas, i.e. it has the simpler form

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

where $n \in \mathbb{N}$. This essentially means that proof steps described by non-applicative inference rules can only make use of properties of the given signature, the predefined predicate symbols '=', 'def' and '<', or the (other) literals occurring in the clause of the goal to be reduced. Our calculus for inductive proofs provides a total of twenty non-applicative inference rules most of which may be easily understood and applied in the construction of inductive proofs.

5.2.1 Establishing Simple Tautologies

Instead of defining *logical* axioms (besides those contained in the underlying admissible specification), our calculus for inductive proofs offers a few very simple inference rules

Complementary Literals

$$\frac{\langle \Gamma, \lambda, \Delta, \lambda', \Pi \,;\, w\rangle}{} \qquad \text{if} \quad \lambda' =_{\text{lit}} \overline{\lambda}.$$

$\neq$-Tautology

$$\frac{\langle \Gamma,\, t_1 \neq t_2 \,, \Delta \,;\, w\rangle}{} \qquad \text{if} \quad \begin{array}{l} - \ t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}}) \\ - \ t_1 \text{ and } t_2 \text{ are not unifiable.} \end{array}$$

$<$-Tautology

$$\frac{\langle \Gamma,\, () < (u_1, \ldots, u_k) \,, \Delta \,;\, w\rangle}{} \qquad \text{if} \quad k > 0.$$

Figure 5.1:

that allow one to prove a goal containing a "tautology", i.e. an obviously inductively valid clause, by reducing it to an *empty* sequence of goals.

The inference rule Complementary Literals (see Figure 5.1) can be used to prove a goal with a clause in which *complementary* literals occur (see Section 3.1.1). Evidently, any clause with complementary literals is (inductively) valid.

The inference rule $\neq$-Tautology may need a more detailed explanation. The rule is based on the requirement that the given specification have free constructors. In this case, any negative equational literal consisting of non-unifiable constructor terms is inductively valid. For instance, the goal $\langle\, 0 \neq \mathsf{s}(x) \ ;\ ()\rangle$, which represents the first of *Peano's postulates* for the natural numbers, may be proved by applying $\neq$-Tautology, since the constructor terms 0 and $\mathsf{s}(x)$ are not unifiable.

The inference rule $<$-Tautology is applicable to goals with order atoms whose left-hand side is the empty tuple. Clearly, any clause containing such an order atom is inductively valid (see Definition 4.2.5 and Section 2.4).

As can be checked easily, the inference rules in Figure 5.1 have the required soundness and safeness properties.

Lemma 5.2.1 *The inference rules* Complementary Literals, $\neq$-Tautology *and* $<$-Tautology *are sound and safe.*

5.2.2 Decomposing Atoms

The reader may have noticed that we still have not presented any inference rules for proving goals with tautological literals of further kinds, such as (i) $t = t$ (ii) $\mathrm{def}(t)$

=-Decomposition

$$\frac{\langle \Gamma,\ t_1 = t_2\,,\ \Delta\,;\ w\rangle}{\langle\, u_1 = v_1\,, \Gamma,\ t_1 = t_2\,,\ \Delta\,;\ w\rangle\ \ldots\ \langle\, u_k = v_k\,, \Gamma,\ t_1 = t_2\,,\ \Delta\,;\ w\rangle}$$

if – $\mathrm{top}(t_1) = \mathrm{top}(t_2)$

– $u_1 = v_1\,, \ldots,\ u_k = v_k$ are exactly those equations in $\{\, t_1/p \dot{=} t_2/p \mid p \in \mathrm{MinDifPos}(t_1, t_2)\,\}$ whose complements do not occur in Γ, Δ.

Figure 5.2:

where $t \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ or (iii) $t_1 < t_2$ where $t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ and $|t_1| < |t_2|$. As will become evident below, goals with atoms of these kinds can be proved by applying one of the following more general inference rules for *decomposing* certain equational, definedness or order atoms.

We begin with the decomposition rule for equational atoms. Let $t_1, t_2, t \in \mathcal{T}(sig, V)$. The set of *minimal difference positions* $\mathrm{MinDifPos}(t_1, t_2)$ of t_1 and t_2 is defined as the set of minimal positions in $\{\, p \in \mathrm{Pos}(t_1) \cap \mathrm{Pos}(t_2) \mid \mathrm{top}(t_1/p) \neq \mathrm{top}(t_2/p)\,\}$. Now if a goal contains an equation $t_1 = t_2$ such that $\mathrm{top}(t_1) = \mathrm{top}(t_2)$ (or $\varepsilon \notin \mathrm{MinDifPos}(t_1, t_2)$) then we can decompose $t_1 = t_2$ with the inference rule =-Decomposition (see Figure 5.2).

It is easily seen that this rule reduces a goal of the form $\langle \Gamma,\ t = t, \Delta\,;\ w\rangle$ to an empty sequence of goals, because $\mathrm{MinDifPos}(t, t) = \emptyset$. Moreover, the inference rule may make use of the literals in Γ, Δ — the so-called *context* of $t_1 = t_2$: For instance, the goal

$$\langle\, x_1 \neq x_2 \ \vee\ \mathsf{plus}(\mathsf{s}(x_1), y_1) = \mathsf{plus}(\mathsf{s}(x_2), y_2)\ \vee\ y_2 \neq y_1\ ;\ ()\rangle$$

can be proved by applying the rule =-Decomposition (i.e. $k = 0$), since the complements of the equations $x_1 = x_2$ and $y_1 = y_2$ occur in the context of the decomposed equation $\mathsf{plus}(\mathsf{s}(x_1), y_1) = \mathsf{plus}(\mathsf{s}(x_2), y_2)$.

Note that =-Decomposition should not be applied "blindly"; in other words, not every possible application of =-Decomposition is actually useful. To see this consider e.g. the goal $\langle\, \mathsf{plus}(x, y) = \mathsf{plus}(y, x)\ ;\ x\rangle$ whose reduction with =-Decomposition yields the two goals $\langle\, x = y\ \vee\ \mathsf{plus}(x, y) = \mathsf{plus}(y, x)\ ;\ x\rangle$ and $\langle\, y = x\ \vee\ \mathsf{plus}(x, y) = \mathsf{plus}(y, x)\ ;\ x\rangle$. Apparently, the problem of proving these two goals is not any easier than the problem of finding a proof for the original goal, as neither of the newly generated equations $x = y$ and $y = x$ is of any use — they both describe the case that $x \neq y$.

Evidently, the safeness of =-Decomposition follows directly from Lemma 5.1.2.

For the presentation of the decomposition rule for definedness atoms we need to introduce yet another set of term positions. The set of *minimal non-constructor positions* $\mathrm{MinNonCPos}(t)$ of a term $t \in \mathcal{T}(sig, V)$ is defined as the set of minimal positions in $\{\, p \in \mathrm{Pos}(t) \mid \mathrm{top}(t/p) \notin (C \cup V^{\mathrm{C}})\,\}$. Now given a goal which contains a definedness def-Decomposition

$$\frac{\langle \Gamma,\ \mathrm{def}(t)\,,\Delta\,;\ w\rangle}{\langle\,\mathrm{def}(u_1)\,,\Gamma,\ \mathrm{def}(t)\,,\Delta\,;\ w\rangle\ \ \ldots\ \ \langle\,\mathrm{def}(u_k)\,,\Gamma,\ \mathrm{def}(t)\,,\Delta\,;\ w\rangle}$$

if – $\mathrm{top}(t) \in (C \cup V^{\mathrm{C}})$

– $\mathrm{def}(u_1)\,,\ldots,\ \mathrm{def}(u_k)$ are exactly those definedness atoms in $\{\,\mathrm{def}(t/p)\ |\, p \in \mathrm{MinNonCPos}(t)\,\}$ whose complements do not occur in Γ, Δ.

Figure 5.3:

atom of the form $\mathrm{def}(t)$ such that $\mathrm{top}(t) \in (C \cup V^{\mathrm{C}})$ (or $\varepsilon \notin \mathrm{MinNonCPos}(t)$), the inference rule def-Decomposition (see Figure 5.3) can be used to decompose $\mathrm{def}(t)$.

Obviously, $\mathrm{MinNonCPos}(t)$ is empty for $t \in \mathcal{T}(sig^C, V^{\mathrm{C}})$. Hence, we can use the rule def-Decomposition to solve a goal of the form $\langle \Gamma,\ \mathrm{def}(t)\,,\Delta\,;\ w\rangle$ if $t \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ (i.e. $k = 0$). As in the case of the preceding inference rule, def-Decomposition also takes the context Γ, Δ of $\mathrm{def}(t)$ into account. For instance, let $X \in V^{\mathrm{G}}$. Then the goal

$$\langle\,\mathrm{def}(\mathsf{cons}(\mathsf{s}(X), \mathsf{app}(l_1, l_2)))\ \vee\ \neg\mathrm{def}(X)\ ;\ l_1\rangle$$

can be reduced with def-Decomposition to the goal

$$\langle\,\mathrm{def}(\mathsf{app}(l_1, l_2))\ \vee\ \mathrm{def}(\mathsf{cons}(\mathsf{s}(X), \mathsf{app}(l_1, l_2)))\ \vee\ \neg\mathrm{def}(X)\ ;\ l_1\rangle$$

(i.e. $k = 1$ although $\mathsf{cons}(\mathsf{s}(X), \mathsf{app}(l_1, l_2))$ has *two* minimal non-constructor positions).

Like =-Decomposition, the inference rule def-Decomposition is trivially safe, because every subgoal of the rule is obtained by adding the literal $\mathrm{def}(u_i)$ to the goal of the rule (see Lemma 5.1.2). However, if the subgoals were of the form $\langle \Gamma,\ \mathrm{def}(u_i)\,,\Delta\,;\ w\rangle$ — instead of $\langle\,\mathrm{def}(u_i)\,,\Gamma,\ \mathrm{def}(t)\,,\Delta\,;\ w\rangle$ — then def-Decomposition would not be a safe inference rule anymore:

Example 5.2.2 Let $spec_{\mathsf{p}} = (sig, C, R)$ be a specification with constructors of the natural numbers where $C = \{\mathsf{0}, \mathsf{s}\}$ and p is a defined operator axiomatized by

$$\begin{aligned}\mathsf{p}(\mathsf{s}(x)) &= x\\ \mathsf{s}(\mathsf{p}(x)) &= x\end{aligned}$$

Then $spec_{\mathsf{p}}$ is an admissible specification with free constructors. Now $\mathrm{def}(\mathsf{s}(\mathsf{p}(\mathsf{0})))$ is inductively valid w.r.t. $spec_{\mathsf{p}}$ but $\mathrm{def}(\mathsf{p}(\mathsf{0}))$ is not. Therefore, any inference rule allowing one to reduce the goal $\langle\,\mathrm{def}(\mathsf{s}(\mathsf{p}(\mathsf{0})))\ ;\ ()\rangle$ to $\langle\,\mathrm{def}(\mathsf{p}(\mathsf{0}))\ ;\ ()\rangle$ cannot be safe.[2]

Lemma 5.2.3 *The inference rules* =-Decomposition *and* def-Decomposition *are sound and safe.*

$<$-Decomposition

$$\frac{\langle \Gamma,\, t_1 < t_2\,,\, \Delta\,;\; w\rangle}{\langle\, \mathrm{def}(u_1)\,,\Gamma,\, t_1 < t_2\,,\, \Delta\,;\; w\rangle \;\;\ldots\;\; \langle\, \mathrm{def}(u_k)\,,\Gamma,\, t_1 < t_2\,,\, \Delta\,;\; w\rangle}$$

if
- $\mathrm{top}(t_2) \in C$
- there are $\hat{t}_1, \hat{t}_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ 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 $x \in V^{\mathrm{C}}$
  - $\mathrm{def}(u_1)\,, \ldots,\, \mathrm{def}(u_k)$ are exactly those definedness atoms in $$\{\, \mathrm{def}(t_i/p) \mid i \in \{1,2\} \wedge p \in \mathrm{MinNonCPos}(t_i)\,\}$$ that do not occur in $\overline{\Gamma}, \overline{\Delta}$.

Figure 5.4:

We now present the decomposition rule for order atoms (see Figure 5.4). For this purpose we introduce a further technical notion, namely that of so-called *C-fronts*. Intuitively seen, a C-front for a term t is a constructor term $\hat{t}$ that is obtained from t by "consistently"[3] replacing the maximal non-constructor sub-terms of t with *new*[4] constructor variables.

Definition 5.2.4 Suppose $sig = (S, F, \alpha)$ is a signature such that $C \subseteq F$ is a set of constructors for sig. A *C-front* for a term $t \in \mathcal{T}(sig, V)_s$ is a term $\hat{t} \in \mathcal{T}(sig^C, V^{\mathrm{C}})_s$ satisfying:

(a) If $\mathrm{top}(t) \in (V^{\mathrm{G}} \cup F \backslash C)$ then $\hat{t} \in V^{\mathrm{C}}$.

(b) If $t \in V^{\mathrm{C}}$ then $\hat{t} = t$.

(c) If $t = c(t_1, \ldots, t_n)$ and $c \in C$ then $\hat{t} = c(\hat{t}_1, \ldots, \hat{t}_n)$ where, for $i = 1, \ldots, n$, $\hat{t}_i$ is a C-front for t_i.

(d) For each $p_1, p_2 \in \mathrm{MinNonCPos}(t)$, $t/p_1 = t/p_2$ iff $\hat{t}/p_1 = \hat{t}/p_2$.

(e) $\mathrm{Var}(t) \cap \{\, \hat{t}/p \mid p \in \mathrm{MinNonCPos}(t)\,\} = \emptyset$.

Assuming that $x, z, l \in V^{\mathrm{C}}$, a C-front for $\mathsf{cons}(\mathsf{s}(x), \mathsf{cons}(\mathsf{plus}(\mathsf{s}(x), y), \mathsf{app}(l_1, l_2)))$ is $\mathsf{cons}(\mathsf{s}(x), \mathsf{cons}(z, l))$. Furthermore, $\mathsf{cons}(x, \mathsf{cons}(x, l))$ is a C-front for the term $\mathsf{cons}(X, \mathsf{cons}(X, l))$ if $X \in V^{\mathrm{G}}$. Note that the C-fronts of a term t are essentially unique — they can only differ in the new constructor variables used to replace the maximal non-constructor sub-terms of t. Therefore, t is the only C-front for t if $t \in \mathcal{T}(sig^C, V^{\mathrm{C}})$.

[2]Note that def-Decomposition reduces $\langle\, \mathrm{def}(\mathsf{s}(\mathsf{p}(0)))\ ;\ ()\rangle$ to $\langle\, \mathrm{def}(\mathsf{p}(0)) \vee \mathrm{def}(\mathsf{s}(\mathsf{p}(0)))\ ;\ ()\rangle$.

[3]see condition (d) in Definition 5.2.4

[4]see condition (e) ib.

As a consequence, any goal containing an order atom that may typically be generated in proofs by structural induction, such as e.g. $x < \mathsf{s}(x)$ or $l < \mathsf{cons}(x, l)$, can be proved with the rule <-Decomposition (i.e. $k = 0$). Furthermore, <-Decomposition can also be used to decompose order atoms not made up exclusively of constructor terms. For instance, an order subgoal of the form

$$\langle\, \mathsf{len}(l) < \mathsf{s}(\mathsf{len}(l)) \;\vee \ldots \;;\; \mathsf{len}(\mathsf{cons}(x, l))\rangle$$

arising in a proof by induction on the length of lists (see Example 4.2.4) can be reduced to

$$\langle\, \mathrm{def}(\mathsf{len}(l)) \;\vee\; \mathsf{len}(l) < \mathsf{s}(\mathsf{len}(l)) \;\vee \ldots \;;\; \mathsf{len}(\mathsf{cons}(x, l))\rangle$$

As in the case of the inference rule def-Decomposition, we refer to the specification $spec_{\mathsf{p}}$ from Example 5.2.2 to illustrate that <-Decomposition would not be a safe inference rule either if its subgoals were of the form $\langle\, \mathrm{def}(u_i)\,, \Gamma, \Delta\,;\, w\rangle$ instead of $\langle\, \mathrm{def}(u_i)\,, \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle$. A reason for this is that the clause $\mathsf{0} < \mathsf{s}(\mathsf{s}(\mathsf{p}(\mathsf{0})))$ is an inductive theorem w.r.t. $spec_{\mathsf{p}}$ whereas the clause $\mathrm{def}(\mathsf{p}(\mathsf{0}))$ is not inductively valid w.r.t. $spec_{\mathsf{p}}$.[5]

Lemma 5.2.5 *The inference rule* <-Decomposition *is sound and safe.*

Again, the safeness of <-Decomposition is an immediate consequence of Lemma 5.1.2.

5.2.3 Removing Redundant Literals

Literals which occur more than once in a goal or which are obviously unsatisfiable, such as e.g. (i) $\mathrm{t} \neq \mathrm{t}$ (ii) $\neg\mathrm{def}(t)$ where $t \in \mathcal{T}(sig^C, V^C)$ or (iii) $\mathsf{0} = \mathsf{s}(x)$, are *redundant* in a goal $\langle \Gamma\,;\, w\rangle$ in the sense that they do not contribute to the inductive validity of Γ. At least from a practical point of view, it may be important to be able to remove such literals from $\langle \Gamma\,;\, w\rangle$ — firstly to concentrate on the relevant information in $\langle \Gamma\,;\, w\rangle$ and secondly to eventually derive the empty clause (in some cases) when a false conjecture is to be proved. For this purpose we provide the following five inference rules (see Figure 5.5).

The inference rule Multiple Literals can be used to remove multiple occurrences of a literal in a goal.

The inference rule =-Removal is provided for eliminating equations from a goal that consist of non-unifiable constructor terms, such as e.g. $\mathsf{0} = \mathsf{s}(x)$. As we require free constructors in the underlying specification any such equation is unsatisfiable.

The inference rule <-Removal is applicable to goals with order atoms whose *right* hand side is the empty tuple. Clearly, such order atoms are unsatisfiable (see Definition 4.2.5 and Section 2.4).

[5] Observe that <-Decomposition reduces $\langle\, \mathsf{0} < \mathsf{s}(\mathsf{s}(\mathsf{p}(\mathsf{0})))\;;\;()\rangle$ to $\langle\, \mathrm{def}(\mathsf{p}(\mathsf{0})) \;\vee\; \mathsf{0} < \mathsf{s}(\mathsf{s}(\mathsf{p}(\mathsf{0})))\;;\;()\rangle$.

Multiple Literals

$$\frac{\langle \Gamma, \lambda, \Delta, \lambda', \Pi \,;\, w\rangle}{\langle \Gamma, \lambda, \Delta, \Pi \,;\, w\rangle} \quad \text{if} \quad \lambda' =_{\mathrm{lit}} \lambda.$$

=-Removal

$$\frac{\langle \Gamma,\ t_1 = t_2\ , \Delta \,;\, w\rangle}{\langle \Gamma, \Delta \,;\, w\rangle} \quad \text{if} \quad \begin{array}{l} - \ \ t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}}) \\ - \ \ t_1 \text{ and } t_2 \text{ are not unifiable.} \end{array}$$

<-Removal

$$\frac{\langle \Gamma,\ (u_1, \ldots, u_k) < ()\ , \Delta \,;\, w\rangle}{\langle \Gamma, \Delta \,;\, w\rangle}$$

≠-Removal

$$\frac{\langle \Gamma,\ t_1 \neq t_2\ , \Delta \,;\, w\rangle}{\langle \Gamma, \Delta \,;\, w\rangle} \quad \text{if} \quad \begin{array}{l} - \ \ \mathrm{top}(t_1) = \mathrm{top}(t_2) \\ - \ \ \text{each atom in } \{\, t_1/p \neq t_2/p \mid p \in \mathrm{MinDifPos}(t_1, t_2) \,\} \\ \quad\ \ \text{occurs in } \Gamma, \Delta. \end{array}$$

¬def-Removal

$$\frac{\langle \Gamma,\ \neg\mathrm{def}(t)\ , \Delta \,;\, w\rangle}{\langle \Gamma, \Delta \,;\, w\rangle} \quad \text{if} \quad \begin{array}{l} - \ \ \mathrm{top}(t) \in (C \cup V^{\mathrm{C}}) \\ - \ \ \text{each atom in } \{\, \neg\mathrm{def}(t/p) \mid p \in \mathrm{MinNonCPos}(t) \,\} \\ \quad\ \ \text{occurs in } \Gamma, \Delta. \end{array}$$

Figure 5.5:

Since $\mathrm{MinDifPos}(t, t) = \emptyset$, we may apply the inference rule ≠-Removal to reduce a goal of the form $\langle \Gamma,\, t \neq t\, , \Delta \,;\, w\rangle$ to the goal $\langle \Gamma, \Delta \,;\, w\rangle$. As in the case of the rule =-Decomposition, ≠-Removal can also make use of the context Γ, Δ of $t_1 \neq t_2$. For example, $\langle\, \mathsf{s}(x) \neq \mathsf{s}(y) \,\vee\, y \neq x \,;\, ()\rangle$ is reduced to $\langle\, y \neq x \,;\, ()\rangle$ in one step.

The inference rule ¬def-Removal reduces any goal of the form $\langle \Gamma,\, \neg\mathrm{def}(t)\, , \Delta \,;\, w\rangle$, where $t \in \mathcal{T}(sig^C, V^{\mathrm{C}})$, to $\langle \Gamma, \Delta \,;\, w\rangle$, for then $\mathrm{MinNonCPos}(t) = \emptyset$. Analogously to the rule def-Decomposition, ¬def-Removal may also utilize the context Γ, Δ of $\neg\mathrm{def}(t)$ (in this case to establish the unsatisfiability of $\neg\mathrm{def}(t)$). For instance, the goal $\langle\, \neg\mathrm{def}(\mathsf{cons}(\mathsf{s}(X), \mathsf{app}(l_1, l_2))) \,\vee\, \neg\mathrm{def}(X) \,\vee\, \neg\mathrm{def}(\mathsf{app}(l_1, l_2)) \,;\, l_1\rangle$ can be reduced with def-Removal to $\langle\, \neg\mathrm{def}(X) \,\vee\, \neg\mathrm{def}(\mathsf{app}(l_1, l_2)) \,;\, l_1\rangle$.

Lemma 5.2.6 *The inference rules* Multiple Literals, =-Removal, <-Removal, ≠-Removal *and* ¬def-Removal *are sound and safe.*

Note that these inference rules are trivially sound.

Constant Rewriting

$$\frac{\langle \Gamma, \lambda[t_1]_p, \Delta\,;\, w\rangle}{\langle \Gamma, \lambda[t_2]_p, \Delta\,;\, w\rangle} \quad \text{if} \quad \begin{array}{l} - \;\; p \in \mathrm{Pos}(\lambda) \text{ and } \lambda/p = t_1 \\ - \;\; \text{there is a literal } t_1 \dot{\neq} t_2 \text{ in } \Gamma, \Delta\,. \end{array}$$

$\neq$-Unification

$$\frac{\langle \Gamma,\, t_1 \neq t_2\,, \Delta\,;\, w\rangle}{\langle \Gamma\tau, \Delta\tau\,;\, w\tau\rangle} \quad \text{if} \quad \begin{array}{l} - \;\; t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}}) \\ - \;\; \tau = \mathrm{mgu}(\,t_1 = t_2\,, \mathrm{Var}(\Gamma, t_1, t_2, \Delta, w)) \text{ exists.} \end{array}$$

Constructor Variable Addition

$$\frac{\langle \Gamma,\, \neg\mathrm{def}(t)\,, \Delta\,;\, w\rangle}{\langle \Gamma,\, x \neq t\,, \Delta\,;\, w\rangle} \quad \text{if} \quad x \in V^{\mathrm{C}} \backslash \mathrm{Var}(\Gamma, \Delta, t, w).$$

Figure 5.6:

5.2.4 Making Use of Negative Literals

A major part of the inference rules of our calculus for inductive proofs make it possible to utilize literals in the *context* Γ, Δ of the literal λ in the goal $\langle \Gamma, \lambda, \Delta\,;\, w\rangle$ in order to generate fewer subgoals than needed otherwise. Examples of such rules are the ones presented in Sections 5.2.2 and 5.3. In what follows we are going to introduce three inference rules that make use of negative (equational or definedness) literals in clauses explicitly and in a rather elementary way (see Figure 5.6).

The inference rule Constant Rewriting utilizes a negative equational literal $t_1 \neq t_2$ to rewrite terms occurring in any *other* literal of the goal with the equation $t_1 = t_2$. However, the variables in $t_1 = t_2$ are treated as *constants*, i.e. no matching substitution may be used. For instance, assume that $X, Y, Z \in V^{\mathrm{G}}$. We can apply the rule Constant Rewriting to the goal $\langle\, X \neq Y \,\vee\, Y \neq Z \,\vee\, X = Z\,;\, ()\rangle$, which expresses the transitivity of '='. By rewriting the term Y in the second literal with the first literal we obtain the (sub-) goal $\langle\, X \neq Y \,\vee\, X \neq Z \,\vee\, X = Z\,;\, ()\rangle$. This goal can be proved using the rule Complementary Literals (see Figure 5.1).

Lemma 5.2.7 *The inference rule* Constant Rewriting *is sound and safe.*

The inference rule $\neq$-Unification is another rule which makes use of the requirement that the constructors be free in the underlying specification. In case the constructor terms t_1 and t_2 are unifiable with mgu τ this information is propagated to *all* the other literals in the goal $\langle \Gamma,\, t_1 \neq t_2\,, \Delta\,;\, w\rangle$ by instantiating them with τ.[6] Consider e.g. the goal $\langle\, \mathsf{s}(x) \neq \mathsf{s}(y) \,\vee\, x = y\,;\, ()\rangle$ which represents the second of Peano's Postulates

[6]Observe that $t_1\tau$ and $t_2\tau$ are identical terms so that the literal $t_1\tau \neq t_2\tau$ is actually redundant.

Tuple $<$-Reduction

$$\frac{\langle \Gamma,\ (t_1,u_1,\ldots,u_m) < (t_2,v_1,\ldots,v_n)\,,\Delta\,;\ w\rangle}{\langle\, t_1 < t_2\,,\Gamma,\ (t_1,u_1,\ldots,u_m) < (t_2,v_1,\ldots,v_n)\,,\Delta\,;\ w\rangle} \qquad \text{if} \quad m>0 \ \vee\ n>0.$$

Tuple $=$-Reduction

$$\frac{\langle \Gamma,\ (t,u_1,\ldots,u_m) < (t,v_1,\ldots,v_n)\,,\Delta\,;\ w\rangle}{\langle \Gamma,\ (u_1,\ldots,u_m) < (v_1,\ldots,v_n)\,,\Delta\,;\ w\rangle}$$

Figure 5.7:

for the natural numbers. Applying $\neq$-Unification to the first literal we obtain the goal $\langle\, y = y\ ;\ ()\rangle$, which can be proved with the rule $=$-Decomposition (see Figure 5.2).

Lemma 5.2.8 *The inference rule* $\neq$-Unification *is sound and safe.*

The inference rule Constructor Variable Addition allows one to replace a negative literal of the form $\neg\mathrm{def}(t)$ with $x \neq t$ where x is a "new" constructor variable (see Figure 5.6). The literal $x \neq t$ can be used then to rewrite occurrences of t with x by applying the rule Constant Rewriting. For example, the goal $\langle\, \mathsf{s}(X) \neq \mathsf{0}\ \vee\ \neg\mathrm{def}(X)\ ;\ ()\rangle$ can be reduced to $\langle\, \mathsf{s}(X) \neq \mathsf{0}\ \vee\ y \neq X\ ;\ ()\rangle$ using Constructor Variable Addition where $X \in V^{\mathrm{G}}_{\mathsf{nat}}$. An ensuing application of Constant Rewriting yields the goal $\langle\, \mathsf{s}(y) \neq \mathsf{0}\ \vee\ y \neq X\ ;\ ()\rangle$, which can be proved with $\neq$-Tautology (see Figure 5.1).

Lemma 5.2.9 *The inference rule* Constructor Variable Addition *is sound and safe.*

5.2.5 Further Inference Rules for Order Atoms

Besides $<$-Tautology, $<$-Decomposition and $<$-Removal, we still have to present four further inference rules applicable to goals with order atoms in order to achieve the required integration of the semantic induction order (as defined in Section 4.2; see also Section 4.3) into our calculus for inductive proofs.

The inference rules Tuple $<$-Reduction and Tuple $=$-Reduction may be applied in cases where (non-trivial) tuple weights are to be compared lexicographically (see Figure 5.7). To illustrate these inference rules we provide a few examples. Let $X, Y, Z_1, Z_2 \in V^{\mathrm{G}}$.

By means of Tuple $<$-Reduction, the goal $\langle\, (x, X) < (\mathsf{s}(x), Y)\ ;\ ()\rangle$ can be reduced to $\langle\, x < \mathsf{s}(x)\ \vee\ (x, X) < (\mathsf{s}(x), Y)\ ;\ ()\rangle$, which is provable with the rule $<$-Decomposition. Moreover, an application of Tuple $<$-Reduction to $\langle\, (X, Z_1) < (Y, Z_2)\ \vee\ X \not< Y\ ;\ ()\rangle$ yields $\langle\, X < Y\ \vee\ (X, Z_1) < (Y, Z_2)\ \vee\ X \not< Y\ ;\ ()\rangle$, which can be proved with the rule Complementary Literals.[7]

[7]Note that both $(x, X) < (\mathsf{s}(x), Y)$ and $X < Y\ \vee\ (X, Z_1) < (Y, Z_2)\ \vee\ X \not< Y$ are inductively valid w.r.t. $\mathit{spec}_{\mathsf{natlist}}$.

<-Monotonicity

$$\frac{\langle \Gamma,\, t_1[u_1]_{p_1} < t_2[u_2]_{p_2} \,, \Delta\,;\, w\rangle}{\langle\, u_1 < u_2 \,, \Gamma,\, t_1[u_1]_{p_1} < t_2[u_2]_{p_2} \,, \Delta\,;\, w\rangle}$$

if – for $i \in \{1,2\}$, $p_i \in \mathrm{Pos}(t_i)\backslash\{\varepsilon\}$ and $t_i/p_i = u_i$
– there are $\hat{t}_1, \hat{t}_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ and $x_1, x_2 \in V^{\mathrm{C}}\backslash\mathrm{Var}(t_1,t_2)$ 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 $x \in V^{\mathrm{C}}\backslash\{x_1,x_2\}$, $|\hat{t}_1|_x \leq |\hat{t}_2|_x$.

<-Transitivity

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

Figure 5.8:

The use of the inference rule Tuple =-Reduction can be exemplified as follows: Given the goal $\langle\, (X, Z_1) < (Y, Z_2) \,\vee\, X \neq Y \,\vee\, Z_1 \not< Z_2 \;;\; ()\rangle$[8] we can apply Constant Rewriting to rewrite the first literal with the second. This gives us $\langle\, (Y, Z_1) < (Y, Z_2) \,\vee\, X \neq Y \,\vee\, Z_1 \not< Z_2 \;;\; ()\rangle$, which can be reduced to $\langle\, Z_1 < Z_2 \,\vee\, X \neq Y \,\vee\, Z_1 \not< Z_2 \;;\; ()\rangle$ using Tuple =-Reduction. This goal is also provable with the rule Complementary Literals.

Lemma 5.2.10 *The inference rules* Tuple <-Reduction *and* Tuple =-Reduction *are sound and safe.*

The two remaining inference rules for goals with order atoms, namely <-Monotonicity and <-Transitivity (see Figure 5.8), are usually needed in the induction steps of proofs of so-called *induction lemmas* for destructors (see Section 4.3).

A typical application of <-Monotonicity is described in the subsequent example.

Example 5.2.11 Suppose the specification $spec_{\mathsf{natlist}}$ is extended with the following defining rules

$$\begin{array}{lcl} \mathsf{split2}(\mathsf{nil}) & = & \mathsf{nil} \\ \mathsf{split2}(\mathsf{cons}(x, \mathsf{nil})) & = & \mathsf{nil} \\ \mathsf{split2}(\mathsf{cons}(x, \mathsf{cons}(y, l))) & = & \mathsf{cons}(y, \mathsf{split2}(l)) \end{array}$$

These rewrite rules axiomatize a function which returns the elements of a list at even positions (beginning with the second element). Together with a similar function split1 for the elements of a list at odd positions, split2 may be used as a destructor in the definition of a mergesort operation on lists.

[8]the clause in which is also inductively valid w.r.t. $spec_{\mathsf{natlist}}$

Apparently the clause $\mathsf{split2}(l) < l \lor l = \mathsf{nil}$ is an induction lemma for split2. In that part of the proof (state graph) for its inductive validity which corresponds to the induction step in an informal proof the goal

$$\langle\, \mathsf{cons}(y, \mathsf{split2}(l)) < \mathsf{cons}(x, \mathsf{cons}(y, l)) \;;\; \mathsf{cons}(x, \mathsf{cons}(y, l))\rangle$$

arises. By applying <-Monotonicity (with $\hat{t}_1 = \mathsf{cons}(y, l_1)$ and $\hat{t}_2 = \mathsf{cons}(x, \mathsf{cons}(y, l_2))$) we can reduce this goal to

$$\langle\, \mathsf{split2}(l) < l \,\lor\, \mathsf{cons}(y, \mathsf{split2}(l)) < \mathsf{cons}(x, \mathsf{cons}(y, l)) \;;\; \mathsf{cons}(x, \mathsf{cons}(y, l))\rangle$$

to which the conjecture $\langle\, \mathsf{split2}(l) < l \lor l = \mathsf{nil} \;;\; l\rangle$ is applicable as induction hypothesis.

Due to Lemma 5.1.2, the inference rule <-Monotonicity is trivially safe, because the subgoal of the rule is obtained by adding the literal $u_1 < u_2$ to the goal of the rule. However, as in the cases of the inference rules def-Decomposition and <-Decomposition, a variant of <-Monotonicity given by the simpler subgoal $\langle\, u_1 < u_2 \,, \Gamma, \Delta \,;\, w\rangle$ (instead of $\langle\, u_1 < u_2 \,, \Gamma,\, t_1[u_1]_{p_1} < t_2[u_2]_{p_2} \,, \Delta \,;\, w\rangle$) is not a safe inference rule: The order atom $\mathsf{s}(0) < \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{p}(0))))$ is inductively valid w.r.t. $\mathit{spec}_{\mathsf{p}}$ from Example 5.2.2, but not the order atom $0 < \mathsf{p}(0)$, which can be inferred by applying this variant of <-Monotonicity.[9]

Lemma 5.2.12 *The inference rule* <-Monotonicity *is sound and safe.*

Like <-Monotonicity, the inference rule <-Transitivity is primarily intended to be used in the induction steps of proofs of induction lemmas. Consider e.g. the induction lemma for the destructor minus, namely the clause $\mathsf{minus}(x, y) < x \lor \mathsf{less}(x, y) = \mathsf{true} \lor y = 0$. A straightforward construction of a proof state graph for the conjecture

$$\langle\, \mathsf{minus}(x, y) < x \,\lor\, \mathsf{less}(x, y) = \mathsf{true} \,\lor\, y = 0 \;;\; x\rangle \tag{5.1}$$

yields the goal

$$\langle\, \mathsf{minus}(x, y) < \mathsf{s}(x) \,\lor\, \mathsf{less}(x, y) = \mathsf{true} \;;\; \mathsf{s}(x)\rangle$$

in the induction step. An application of the rule <-Transitivity (with $w_2 = x$) results in the two goals

$$\langle\, \mathsf{minus}(x, y) < x \,\lor\, \mathsf{minus}(x, y) < \mathsf{s}(x) \,\lor\, \mathsf{less}(x, y) = \mathsf{true} \;;\; \mathsf{s}(x)\rangle$$

$$\langle\, x < \mathsf{s}(x) \,\lor\, \mathsf{minus}(x, y) < \mathsf{s}(x) \,\lor\, \mathsf{less}(x, y) = \mathsf{true} \;;\; \mathsf{s}(x)\rangle$$

The first of these goals can be "simplified" with the conjecture (5.1) as induction hypothesis, whereas the second goal is provable with <-Decomposition (see Figure 5.4).

Lemma 5.2.13 *The inference rule* <-Transitivity *is sound and safe.*

[9] <-Monotonicity reduces $\langle\, \mathsf{s}(0) < \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{p}(0)))) \;;\; ()\rangle$ to $\langle\, 0 < \mathsf{p}(0) \,\lor\, \mathsf{s}(0) < \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{p}(0)))) \;;\; ()\rangle$.

Substitution Addition

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

Figure 5.9:

5.2.6 Non-Applicative Case Analyses

The two remaining *non*-applicative inference rules yet to be introduced allow one to generate sequences of subgoals that describe case analyses for the goals these rules are applied to.

In the case of the inference rule Substitution Addition (see Figure 5.9) the case analysis is determined by a certain set of constructor substitutions, namely a so-called *cover set of substitutions*. Roughly speaking, the substitutions in a cover set for a goal $\langle \Gamma \,;\, w\rangle$ are intended to instantiate certain (constructor) variables in Γ with constructor terms to yield the goals that represent the various cases in a proof by structural induction.

Definition 5.2.14 A set $\{\sigma_1,\ldots,\sigma_n\}$ of constructor substitutions is called 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 for each $x \in \mathrm{Var}(\Gamma, w)$, $x\sigma = x\sigma_j\tau$.

For example, if $\sigma_1 = \{x \leftarrow \mathsf{0}\}$ and $\sigma_2 = \{x \leftarrow \mathsf{s}(x)\}$ then $\{\sigma_1, \sigma_2\}$ is obviously a cover set of substitutions for the goal $\langle\, \mathsf{less}(x, \mathsf{s}(x)) = \mathsf{true} \,;\, x\rangle$. Thus, applying Substitution Addition with the cover set of substitutions $\{\sigma_1, \sigma_2\}$ gives rise to $\langle\, \mathsf{less}(\mathsf{0}, \mathsf{s}(\mathsf{0})) = \mathsf{true} \,;\, \mathsf{0}\rangle$ (*base case* goal) and $\langle\, \mathsf{less}(\mathsf{s}(x), \mathsf{s}(\mathsf{s}(x))) = \mathsf{true} \,;\, \mathsf{s}(x)\rangle$ (*induction step* goal). As another example consider the goal $\langle\, \mathsf{less}(x, y) \neq \mathsf{true} \vee \mathsf{less}(y, z) \neq \mathsf{true} \vee \mathsf{less}(x, z) = \mathsf{true} \,;\, x\rangle$ which states the transitivity of less. A cover set of substitutions that leads to *easily* provable subgoals consists of the substitutions $\sigma_1 = \{y \leftarrow \mathsf{0}\}$, $\sigma_2 = \{z \leftarrow \mathsf{0}\}$, $\sigma_3 = \{z \leftarrow \mathsf{s}(z), x \leftarrow \mathsf{0}\}$ and $\sigma_4 = \{x \leftarrow \mathsf{s}(x), y \leftarrow \mathsf{s}(y), z \leftarrow \mathsf{s}(z)\}$.

Lemma 5.2.15 *The inference rule* Substitution Addition *is sound and safe.*

The inference rule Literal Addition (see Figure 5.10) can be applied to bring about another kind of case analysis, which is determined by a sequence of literals.

Definition 5.2.16 Let $\lambda_1,\ldots,\lambda_n$ be a clause ($n \in \mathbb{N}$). We define the *case analysis resulting from* $\lambda_1,\ldots,\lambda_n$ to consist of the clauses $\Lambda_1,\ldots,\Lambda_n$ and Λ such that

(a) $\Lambda_i = \overline{\lambda_i}, \lambda_{i-1}, \lambda_{i-2}, \ldots, \lambda_1 \quad \text{for } i = 1,\ldots,n$

(b) $\Lambda = \lambda_n, \ldots, \lambda_1$

Literal Addition

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

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

Figure 5.10:

Let us illustrate this definition and the inference rule Literal Addition with the following example. Assume we are to prove inductive validity of the goal $\langle\, \mathrm{def}(\mathsf{div}(x, \mathsf{s}(y))) \,;\, x \rangle$. Because of the (destructor) recursive axiomatization of div it is appropriate to begin the proof with a case analysis given by the (instantiated) condition literals occurring in the defining rules for div (see Example 2.2.1). That is, an application of Literal Addition with $\lambda_1 = (\, \mathsf{s}(y) = \mathsf{0} \,)$ and $\lambda_2 = (\, \mathsf{less}(x, \mathsf{s}(y)) = \mathsf{true} \,)$ reduces the above conjecture to the three goals

$$\langle\, \mathsf{s}(y) \neq \mathsf{0} \,\vee\, \mathrm{def}(\mathsf{div}(x, \mathsf{s}(y))) \,;\, x \rangle$$

$$\langle\, \mathsf{less}(x, \mathsf{s}(y)) \neq \mathsf{true} \,\vee\, \mathsf{s}(y) = \mathsf{0} \,\vee\, \mathrm{def}(\mathsf{div}(x, \mathsf{s}(y))) \,;\, x \rangle$$

$$\langle\, \mathsf{less}(x, \mathsf{s}(y)) = \mathsf{true} \,\vee\, \mathsf{s}(y) = \mathsf{0} \,\vee\, \mathrm{def}(\mathsf{div}(x, \mathsf{s}(y))) \,;\, x \rangle$$

the first of which is solved with $\neq$-Tautology (see Figure 5.1), while the third literal in the two remaining goals can each be directly rewritten with one of the defining rules for div using the *applicative* inference rule Non-Inductive Rewriting (see Figure 5.12).

Lemma 5.2.17 *The inference rule* Literal Addition *is sound and safe.*

In general there is no straightforward method of deciding whether — and if so, how — to apply either of the inference rules Substitution Addition or Literal Addition to a given goal, as the two foregoing examples may suggest. Another problem posed by the inference rule Substitution Addition is that of ensuring that a set of substitutions is indeed a cover set for a goal. However, it is not difficult to develop sufficient conditions for guaranteeing the cover set property of a set of substitutions which are easily verifiable in most practically relevant cases.

5.3 Applicative Inference Rules

As opposed to the non-applicative inference rules presented so far, each of the remaining five *applicative* inference rules requires one additional goal which is to be applied to the goal of the inference rule either non-inductively (i.e. as an axiom or lemma) or as an induction hypothesis. Before we can deal with these inference rules below, we first have to introduce the notion of the definedness conditions of a (matching) substitution.

Due to specifications with incomplete or non-terminating axiomatizations of partial operations, there may be instances of inductive theorems which are not inductively valid: Consider e.g. the admissible specification $spec_{\mathsf{div}}$ from Example 2.2.1. Clearly, the equation $\mathsf{less}(x, \mathsf{s}(x)) = \mathsf{true}$ is an inductive theorem w.r.t. $spec_{\mathsf{div}}$, but given that $\mu = \{x \leftarrow \mathsf{minus}(0, \mathsf{s}(0))\}$ the instance $\mathsf{less}(x\mu, \mathsf{s}(x\mu)) = \mathsf{true}$ is not even valid in $\mathcal{M}(spec_{\mathsf{div}})$. The problem is that substitutions may assign "undefined" terms to constructor variables. However, if every term $x\mu$ assigned to a constructor variable x by μ is either in $\mathcal{T}(sig^C, V^C)$ or has a so-called *definedness condition*, then the μ-instance of an inductive theorem is in fact an inductive theorem (see Lemma 5.3.2 below).

Definition 5.3.1 The set of *definedness conditions* of a substitution μ and a clause Γ is defined as

$$\mathrm{DefCond}(\mu, \Gamma) = \{\, \neg\mathrm{def}(x\mu) \mid x \in \mathrm{Var}(\Gamma) \cap V^C \wedge x\mu \notin \mathcal{T}(sig^C, V^C)\}$$

Note that $\mathrm{DefCond}(\mu, \Gamma)$ is always finite. Moreover, if μ is a constructor substitution then $\mathrm{DefCond}(\mu, \Gamma) = \emptyset$ for any clause Γ. In order to simplify the presentation of the remaining inference rules, we also denote by $\mathrm{DefCond}(\mu, \Gamma)$ any *sequence* of literals (i.e. a clause) that comprises exactly the literals contained in $\mathrm{DefCond}(\mu, \Gamma)$. This conceptual inaccuracy does not cause any problems, because whenever $\mathrm{DefCond}(\mu, \Gamma)$ is used as a sequence (and not as a set), the order of the literals in the sequence denoted by $\mathrm{DefCond}(\mu, \Gamma)$ is irrelevant.

Lemma 5.3.2 *Suppose $\mathcal{A}$ is a sig-algebra and Γ a clause such that $\mathcal{A} \models \Gamma$. Let μ be a substitution. Then $\mathcal{A} \models \Gamma\mu, \mathrm{DefCond}(\mu, \Gamma)$.*

5.3.1 Non-Inductive Applicative Inference Rules

Each of the non-inductive applicative inference rules of our calculus for inductive proofs is of the form

$$\frac{\langle \Gamma\,;\, w\rangle}{\langle \Gamma_1\,;\, w\rangle \ \ldots\ \langle \Gamma_n\,;\, w\rangle} \quad \text{with} \quad \langle \Pi\,;\, \hat{w}\rangle^{\mathcal{L}}$$

where Π is (the clause representation of) a defining rule of the given specification or another clause to be applied to $\langle \Gamma\,;\, w\rangle$ as a lemma. We call such an inference rule *non-inductive* because $\langle \Pi\,;\, \hat{w}\rangle$ is applied to $\langle \Gamma\,;\, w\rangle$ non-inductively in the sense that $\langle \Pi\,;\, \hat{w}\rangle$ is *not* used as an induction hypothesis, i.e. no order subgoal is generated (see Section 5.3.2).

As is suggested by its name, the inference rule Non-Inductive Subsumption is based on subsumption. It is depicted in Figure 5.11.

Let us first deal with the simplest form of applying Non-Inductive Subsumption. Suppose that the goal $\langle \Pi\,;\, \hat{w}\rangle$ *subsumes* the goal $\langle \Gamma\,;\, w\rangle$, i.e. there is a substitution μ such that Γ contains $\Pi\mu$ (see Section 5.1). In addition, let μ be a constructor substitution. Then $\mathrm{DefCond}(\mu, \Pi)$ is the empty sequence. Hence it is possible to apply Non-Inductive Subsumption to the goal $\langle \Gamma\,;\, w\rangle$ and prove it with the axiom or lemma $\langle \Pi\,;\, \hat{w}\rangle$, as Θ can be chosen to be empty (i.e. $n = 0$).

Non-Inductive Subsumption

$$\frac{\langle \Gamma \,;\, w\rangle}{\langle \Lambda_1, \Gamma \,;\, w\rangle \;\ldots\; \langle \Lambda_n, \Gamma \,;\, w\rangle} \quad \text{with } \langle \Pi \,;\, \hat{w}\rangle^{\mathcal{L}}$$

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

- Γ, Θ contains $\mathrm{DefCond}(\mu, \Pi), \Pi\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.[10]

Figure 5.11:

More complicated applications of the inference rule Non-Inductive Subsumption arise when (1) μ is not a constructor substitution or (2) $\langle \Pi \,;\, \hat{w}\rangle$ subsumes $\langle \Gamma \,;\, w\rangle$ only *partially* in the sense that Γ does not contain $\Pi\mu$ completely. In the general case, an application of Non-Inductive Subsumption to $\langle \Gamma \,;\, w\rangle$ with the axiom or lemma $\langle \Pi \,;\, \hat{w}\rangle$ and the match μ leads to one new subgoal for each literal in $\mathrm{DefCond}(\mu, \Pi)$ or $\Pi\mu$ that does not occur in Γ. These "missing" literals make up the clause Θ (see Figure 5.11).

For instance, let $b \in V^{\mathrm{C}}_{\mathsf{bool}}$ and let $B \in V^{\mathrm{G}}_{\mathsf{bool}}$. Then the goal

$$\langle\, B = \mathsf{true} \,\vee\, B = \mathsf{false} \,\vee\, \neg\mathrm{def}(B) \;;\, ()\rangle$$

can be directly proved by applying Non-Inductive Subsumption with the lemma

$$\langle\, b = \mathsf{true} \,\vee\, b = \mathsf{false} \;;\, ()\rangle$$

although $\mu = \{b \leftarrow B\}$ is not a constructor substitution. To be more precise, we have $\mathrm{DefCond}(\mu,\, b = \mathsf{true} \vee b = \mathsf{false}) = \neg\mathrm{def}(B)$, but as $\neg\mathrm{def}(B)$ occurs in the goal to be reduced, Θ can be empty (i.e. $n = 0$). As another example consider the goal

$$\langle\, \mathsf{less}(0, \mathsf{len}(l)) = \mathsf{true} \,\vee\, \mathsf{len}(l) = 0 \;;\, l\rangle \tag{5.2}$$

An application of Non-Inductive Subsumption with the lemma

$$\langle\, x = y \,\vee\, \mathsf{less}(x, y) = \mathsf{true} \,\vee\, \mathsf{less}(y, x) = \mathsf{true} \;;\, x\rangle \tag{5.3}$$

reduces (5.2) to the two goals

$$\langle\, \mathrm{def}(\mathsf{len}(l)) \,\vee\, \mathsf{less}(0, \mathsf{len}(l)) = \mathsf{true} \,\vee\, \mathsf{len}(l) = 0 \;;\, l\rangle \tag{5.4}$$

$$\langle\, \mathsf{less}(\mathsf{len}(l), 0) \neq \mathsf{true} \,\vee\, \neg\mathrm{def}(\mathsf{len}(l)) \,\vee\, \mathsf{less}(0, \mathsf{len}(l)) = \mathsf{true} \,\vee\, \mathsf{len}(l) = 0 \;;\, l\rangle \tag{5.5}$$

Here, $\mu = \{x \leftarrow \mathsf{len}(l),\, y \leftarrow 0\}$ so that one definedness condition for μ and the lemma is needed, namely $\neg\mathrm{def}(\mathsf{len}(l))$. This accounts for the subgoal (5.4), as $\neg\mathrm{def}(\mathsf{len}(l))$ does not occur in (5.2). Besides, the μ-instance of the lemma in (5.3) subsumes (5.2) only partially — (5.2) does not contain the literal $\mathsf{less}(\mathsf{len}(l), 0) = \mathsf{true}$. This gives rise

[10] For the case analysis resulting from a clause refer to Definition 5.2.16.

Non-Inductive Rewriting

$$\frac{\langle \Gamma, \lambda, \Delta \,;\, w\rangle}{\langle \Lambda_1, \Gamma, \lambda, \Delta \,;\, w\rangle \;\ldots\; \langle \Lambda_n, \Gamma, \lambda, \Delta \,;\, w\rangle \;\langle \Lambda, \Gamma, \lambda[r\mu]_p, \Delta \,;\, w\rangle} \quad \text{with } \langle \Pi,\, l \doteq r\,, \Sigma \,;\, \hat{w}\rangle^{\mathcal{L}}$$

if there is a position $p \in \mathrm{Pos}(\lambda)$, a substitution μ and a clause Θ such that

- $\lambda/p = l\mu$
- $\Gamma, \Delta,\, l\mu = r\mu\,, \Theta$ contains $\mathrm{DefCond}(\mu, (\Pi,\, l \doteq r\,, \Sigma)), \Pi\mu, \Sigma\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

Figure 5.12:

to the second subgoal (5.5).[11] Note that (5.4) can be proved by a further application of Non-Inductive Subsumption with the lemma $\langle\, \mathrm{def}(\mathsf{len}(l))\ ;\ l\rangle$, whereas (5.5) can be simplified with the defining rule $\mathsf{less}(x, 0) = \mathsf{false}$ using the inference rule Non-Inductive Rewriting (see below).

Lemma 5.3.3 *The inference rule* Non-Inductive Subsumption *is sound and safe.*

For rewriting with defining rules (i.e. axioms) of the given specification or with other clauses to be used as lemmas, our calculus for inductive proofs provides the inference rule Non-Inductive Rewriting (see Figure 5.12). This inference rule allows one to perform a *single* rewrite step to any term λ/p in the clause of the goal $\langle \Gamma, \lambda, \Delta\,;\, w\rangle$ with any equational atom $l \doteq r$ of the axiom or lemma contained in the goal $\langle \Pi,\, l \doteq r\,, \Sigma\,;\, \hat{w}\rangle$. Observe that the clause $\Pi,\, l \doteq r\,, \Sigma$ corresponds to the conditional equation or *rewrite rule* $l \doteq r \leftarrow \overline{\Pi, \Sigma}$. We call $l \doteq r$ the head of the rewrite rule, while the literals in $\overline{\Pi, \Sigma}$ are the *condition literals* of the rewrite rule. Evidently, the rewrite relation $\longrightarrow_R$ associated with the given specification $spec = (sig, C, R)$ is not suited as a basis for rewriting terms in proofs: $\longrightarrow_R$ is defined on $\mathcal{T}(sig, V^{\mathrm{G}})$ only and requires the rewrite rules to be constructor rules or defining rules (see Section 3.2.1), thus excluding many clauses typically needed as lemmas.

In the simplest case, there are no condition literals, i.e. Π and Σ are empty, and the definedness conditions $\mathrm{DefCond}(\mu,\, l \doteq r\,)$ of the matching substitution μ and $l \doteq r$ occur in Γ, Δ. Then an application of Non-Inductive Rewriting to $\langle \Gamma, \lambda, \Delta\,;\, w\rangle$ with the axiom or lemma $\langle\, l \doteq r\ ;\ \hat{w}\rangle$ leads to one goal, namely $\langle \Gamma, \lambda[r\mu]_p, \Delta\,;\, w\rangle$, i.e. Θ can be empty $(n = 0)$.

For instance, we can reduce the goal (5.5) with the goal $\langle\, \mathsf{less}(x, 0) = \mathsf{false}\ ;\ ()\rangle$, which contains the first defining rule for less (see $spec_{\mathsf{div}}$ in Example 2.2.1), to

$$\langle\, \mathsf{false} \neq \mathsf{true} \,\vee\, \neg\mathrm{def}(\mathsf{len}(l)) \,\vee\, \mathsf{less}(0, \mathsf{len}(l)) = \mathsf{true} \,\vee\, \mathsf{len}(l) = 0\ ;\ l\rangle \tag{5.6}$$

This use of Non-Inductive Rewriting with $n = 0$ is allowed indeed, because the applied defining rule is unconditional and $\mathrm{DefCond}(\mu,\, \mathsf{less}(x, 0) = \mathsf{false}\,) = \neg\mathrm{def}(\mathsf{len}(l))$ occurs

[11] i.e. $\Theta = \neg\mathrm{def}(\mathsf{len}(l)) \vee \mathsf{less}(\mathsf{len}(l), 0) = \mathsf{true} \quad (n = 2)$

in the goal (5.5). Observe that the rewritten goal (5.6) can be directly proved with the inference rule $\neq$-Tautology (see Figure 5.1).

In the general case, applying the inference rule Non-Inductive Rewriting to the goal $\langle \Gamma, \lambda, \Delta\,;\, w\rangle$ with the axiom or lemma $\langle \Pi,\ l \doteq r\,, \Sigma\,;\, \hat{w}\rangle$ and the match μ yields (i) one additional goal for each literal in the definedness conditions $\mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma))$ not occurring in Γ, Δ and (ii) one additional goal for each (negated) condition literal in $\Pi\mu, \Sigma\mu$ of the rewrite rule that does not occur in Γ, Δ.

For example, consider the goal

$$\langle\, y = \mathsf{0} \ \vee\ \mathsf{div1}(\mathsf{times}(y, z), y, \mathsf{0}, \mathsf{0}) = z\ ;\ ()\rangle \tag{5.7}$$

The clause in (5.7) expresses a key property of the non-terminating operation div1 and is inductively valid w.r.t. $spec_{\mathsf{div1}}$ from Example 2.2.2. Experience shows that for a proof of goal (5.7) an adequate computational "invariant" for div1 is helpful, such as (the clause in) the goal

$$\langle\, y = \mathsf{0} \vee \mathsf{less}(x, \mathsf{times}(y, z)) = \mathsf{true} \vee \mathsf{div1}(x, y, \mathsf{0}, \mathsf{0}) = \mathsf{div1}(x, y, z, \mathsf{times}(y, z))\ ;\ z\rangle \tag{5.8}$$

Now Non-Inductive Rewriting enables us to rewrite the term $\mathsf{div1}(\mathsf{times}(y, z), y, \mathsf{0}, \mathsf{0})$ in (5.7) with (i) the lemma (5.8) using the third literal in (5.8) as the head of the rewrite rule and (ii) the match $\mu = \{x \leftarrow \mathsf{times}(y, z)\}$. Accordingly this application of Non-Inductive Rewriting results in the two additional goals (5.9) and (5.10)

$$\langle\, \mathrm{def}(\mathsf{times}(y, z))\ \vee\ y = \mathsf{0}\ \vee\ \mathsf{div1}(\mathsf{times}(y, z), y, \mathsf{0}, \mathsf{0}) = z\ ;\ ()\rangle \tag{5.9}$$

$$\begin{aligned}\langle\, \mathsf{less}(\mathsf{times}(y, z), \mathsf{times}(y, z)) \neq \mathsf{true}\ \vee\ \neg\mathrm{def}(\mathsf{times}(y, z))\ \vee\ y = \mathsf{0}\\ \vee\, \mathsf{div1}(\mathsf{times}(y, z), y, \mathsf{0}, \mathsf{0}) = z\ ;\ ()\rangle\end{aligned} \tag{5.10}$$

$$\begin{aligned}\langle\, \mathsf{less}(\mathsf{times}(y, z), \mathsf{times}(y, z)) = \mathsf{true}\ \vee\ \neg\mathrm{def}(\mathsf{times}(y, z))\ \vee\ y = \mathsf{0}\\ \vee\, \mathsf{div1}(\mathsf{times}(y, z), y, z, \mathsf{times}(y, z)) = z\ ;\ ()\rangle\end{aligned} \tag{5.11}$$

besides the goal (5.11) which contains the rewritten term, since neither the definedness condition $\neg\mathrm{def}(\mathsf{times}(y, z))$ occurs in (5.7) nor the μ-instantiated and negated condition literal $\mathsf{less}(\mathsf{times}(y, z), \mathsf{times}(y, z)) = \mathsf{true}$. Note that the goals (5.9), (5.10) and (5.11) pose relatively simple proof problems.

The foregoing example may convey the idea of how (non-inductive) conditional rewriting of terms in goals with clauses can be done within our calculus for inductive proofs. Given that $\lambda/p = l\mu$ for a $p \in \mathrm{Pos}(\lambda)$ and a substitution μ, rewriting the goal $\langle \Gamma, \lambda, \Delta\,;\, w\rangle$ with the rewrite rule $l = r\ \leftarrow \overline{\Pi, \Sigma}$ contained in the goal $\langle \Pi,\ \underline{l = r}\,, \Sigma\,;\, \hat{w}\rangle$ is *always* possible — regardless of whether or not the condition literals $\overline{\Pi\mu, \Sigma\mu}$ can actually be shown to be satisfied. However, for each condition literal $\lambda'\mu$ not occurring in the context Γ, Δ of λ, a new *condition subgoal* of the form $\langle \lambda'\mu, \Lambda', \Gamma, \lambda, \Delta\,;\, w\rangle$ is

Applicative Literal Removal

$$\frac{\langle \Gamma, \lambda, \Delta\,;\, w\rangle}{\langle \Gamma, \Delta\,;\, w\rangle} \quad \text{with} \quad \langle \Pi, \lambda', \Sigma\,;\, \hat{w}\rangle^{\mathcal{L}}$$

if there is a substitution μ such that

- $\lambda =_{\text{lit}} \overline{\lambda'\mu}$
- Γ, Δ contains $\text{DefCond}(\mu, (\Pi, \lambda', \Sigma)), \Pi\mu, \Sigma\mu$.

Figure 5.13:

generated. A key consequence — and a major advantage — of this delayed or *lazy* condition testing is that the whole system of inference rules is available for the verification of these condition subgoals. On the other hand, there is a certain drawback of the lazy condition testing: The inference rule Non-Inductive Rewriting needs to be applied with some care, i.e. not "blindly". Suppose e.g. we are given the goal

$$\langle\, \mathsf{div}(x,x) = \mathsf{s}(0) \;\vee\; x = 0 \;;\; x\rangle \tag{5.12}$$

which contains an inductive theorem w.r.t. $spec_{\mathsf{div}}$ from Example 2.2.1. With the goal

$$\langle\, \mathsf{div}(x,y) = 0 \;\vee\; y = 0 \;\vee\; \mathsf{less}(x,y) \neq \mathsf{true} \;;\; ()\rangle$$

for the first defining rule for div and $\mu = \{y \leftarrow x\}$ we may — *but should not* — reduce (5.12) to the two goals

$$\langle\, \mathsf{less}(x,x) = \mathsf{true} \;\vee\; \mathsf{div}(x,x) = \mathsf{s}(0) \;\vee\; x = 0 \;;\; x\rangle \tag{5.13}$$

$$\langle\, \mathsf{less}(x,x) \neq \mathsf{true} \;\vee\; 0 = \mathsf{s}(0) \;\vee\; x = 0 \;;\; x\rangle$$

Evidently, the first literal in (5.13) is "false" and hence *redundant*, and by an application of the inference rule Applicative Literal Removal (see below) with the lemma $\langle\, \mathsf{less}(x,x) \neq \mathsf{true} \;;\; x\rangle$ we can in fact remove it from (5.13). As a consequence, we obtain the original conjecture (5.12) again. In other words, rewriting a term with a rewrite rule whose condition literals cannot be verified should be avoided.

Lemma 5.3.4 *The inference rule* Non-Inductive Rewriting *is sound and safe.*

Besides the five inference rules presented in 5.2.3 we provide a further non-inductive and applicative inference rule for removing redundant literals, namely Applicative Literal Removal. This rule can be used to remove a literal from a goal whose redundancy follows from the axiom or lemma applied to the goal (see Figure 5.13). For a further example besides the one given above, let $B \in V^{\mathrm{G}}_{\mathsf{bool}}$. By an application of Applicative Literal Removal we can (safely) remove the redundant literal $B = \mathsf{false}$ from the goal $\langle\, \mathrm{def}(B) \;\vee\; B = \mathsf{false} \;;\; ()\rangle$ with the lemma $\langle\, \mathrm{def}(B) \;\vee\; B \neq \mathsf{false} \;;\; ()\rangle$.

Lemma 5.3.5 *The inference rule* Applicative Literal Removal *is sound and safe.*

Observe that this inference rule is trivially sound.

Inductive Subsumption

$$\frac{\langle \Gamma \,;\, w\rangle}{\langle \Lambda_1, \Gamma \,;\, w\rangle \ \ldots \ \langle \Lambda_n, \Gamma \,;\, w\rangle \ \langle\, \hat{w}\mu < w \,, \Lambda, \Gamma \,;\, w\rangle} \quad \text{with} \ \ \langle \Pi \,;\, \hat{w}\rangle^{\mathcal{I}}$$

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

- Γ, Θ contains $\mathrm{DefCond}(\mu, \Pi), \Pi\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ.

Figure 5.14:

5.3.2 Inductive Applicative Inference Rules

We still have to introduce the two *inductive* applicative inference rules of our calculus for inductive proofs, namely Inductive Subsumption and Inductive Rewriting. In essence, these rules differ from their non-inductive versions Non-Inductive Subsumption and Non-Inductive Rewriting only in that an application of an inductive inference rule results in one additional (sub-) goal as compared to the corresponding non-inductive rule. The general form of our inductive applicative inference rules is

$$\frac{\langle \Gamma \,;\, w\rangle}{\langle \Gamma_1 \,;\, w\rangle \ \ldots \ \langle \Gamma_n \,;\, w\rangle \ \langle\, \hat{w}\mu < w \,, \Gamma_{n+1} \,;\, w\rangle} \quad \text{with} \ \ \langle \Pi \,;\, \hat{w}\rangle^{\mathcal{I}}$$

where $\langle \Pi \,;\, \hat{w}\rangle$ is the goal to be applied as induction hypothesis with μ as the matching substitution. We refer to the subgoal $\langle\, \hat{w}\mu < w \,, \Gamma_{n+1} \,;\, w\rangle$ as the *order subgoal*, because its purpose is to represent the order condition corresponding to the application of the induction hypothesis $\langle \Pi \,;\, \hat{w}\rangle$ to $\langle \Gamma \,;\, w\rangle$ and introduce it as a proof obligation (see also Section 4.3).

Due to the strong resemblance between the inference rules Inductive Subsumption (see Figure 5.14) and Non-Inductive Subsumption we just give a simple example of a typical application of Inductive Subsumption. In the induction step of the proof of the conjecture $\langle\, \mathrm{def}(\mathsf{ack}(x, y)) \,;\, (x, y)\rangle$ the goal $\langle\, \mathrm{def}(\mathsf{ack}(x, \mathsf{ack}(\mathsf{s}(x), y))) \,;\, (\mathsf{s}(x), \mathsf{s}(y))\rangle$ arises (see Example 4.2.3). Now Inductive Subsumption allows one to reduce this goal to

$$\langle\, \mathrm{def}(\mathsf{ack}(\mathsf{s}(x), y)) \ \vee\ \mathrm{def}(\mathsf{ack}(x, \mathsf{ack}(\mathsf{s}(x), y))) \ ;\ (\mathsf{s}(x), \mathsf{s}(y))\rangle \tag{5.14}$$

$$\begin{aligned} \langle\, (x, \mathsf{ack}(\mathsf{s}(x), y)) < (\mathsf{s}(x), \mathsf{s}(y)) \ &\vee\ \neg\mathrm{def}(\mathsf{ack}(\mathsf{s}(x), y)) \\ &\vee \mathrm{def}(\mathsf{ack}(x, \mathsf{ack}(\mathsf{s}(x), y))) \ ;\ (\mathsf{s}(x), \mathsf{s}(y))\rangle \end{aligned} \tag{5.15}$$

by applying the conjecture itself as induction hypothesis with $\mu = \{y \leftarrow \mathsf{ack}(\mathsf{s}(x), y)\}$. Note that the order subgoal (5.15) can be easily solved with consecutive applications of the inference rules Tuple $<$-Reduction and $<$-Decomposition (see 5.2.5).

Lemma 5.3.6 *The inference rule* Inductive Subsumption *is sound and safe.*

Inductive Rewriting

$$\frac{\langle \Gamma, \lambda, \Delta\,;\, w\rangle}{\langle \Lambda_1, \Gamma, \lambda, \Delta\,;\, w\rangle \;\ldots\; \langle \Lambda_n, \Gamma, \lambda, \Delta\,;\, w\rangle \; \langle \Lambda, \Gamma, \lambda[r\mu]_p, \Delta\,;\, w\rangle \; \langle\, \hat{w}\mu < w\,, \Lambda', \Gamma, \lambda, \Delta\,;\, w\rangle}$$

with $\langle \Pi,\ l \doteq r\,, \Sigma\,;\, \hat{w}\rangle^{\mathcal{I}}$

if there is a position $p \in \mathrm{Pos}(\lambda)$, a substitution μ and a clause Θ such that

- $\lambda/p = l\mu$
- $\Gamma, \Delta,\ l\mu = r\mu\,, \Theta$ contains $\mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma)), \Pi\mu, \Sigma\mu$
- $\Lambda_1, \ldots, \Lambda_n, \Lambda$ is the case analysis resulting from Θ
- $\Lambda' = \ l\mu = r\mu\,, \Lambda\,.$

Figure 5.15:

In order to illustrate the inference rule Inductive Rewriting (see Figure 5.15) we also just describe a simple example application. For a better understanding of this rule the reader should refer to the characterization of the corresponding non-inductive inference rule Non-Inductive Rewriting in Section 5.3.1. Consider the goal

$$\langle\, \mathsf{s}(x) = \mathsf{s}(y) \ \vee\ \mathsf{less}(x,y) = \mathsf{true} \ \vee\ \mathsf{less}(y,x) = \mathsf{true}\ ;\ \mathsf{s}(x)\rangle \tag{5.16}$$

which may be generated in the induction step of a typical proof by structural induction for the conjecture $\langle\, x = y \ \vee\ \mathsf{less}(x,y) = \mathsf{true} \ \vee\ \mathsf{less}(y,x) = \mathsf{true}\ ;\ x\rangle$. An application of Inductive Rewriting to (5.16) with the conjecture as induction hypothesis and $\mu = \mathrm{id}_V$ yields the two goals

$$\langle\, \mathsf{s}(y) = \mathsf{s}(y) \ \vee\ \mathsf{less}(x,y) = \mathsf{true} \ \vee\ \mathsf{less}(y,x) = \mathsf{true}\ ;\ \mathsf{s}(x)\rangle$$

$$\langle\, x < \mathsf{s}(x) \ \vee\ y = x \ \vee\ \mathsf{s}(x) = \mathsf{s}(y) \ \vee\ \mathsf{less}(x,y) = \mathsf{true} \ \vee\ \mathsf{less}(y,x) = \mathsf{true}\ ;\ \mathsf{s}(x)\rangle$$

both of which can be proved directly with the inference rules =-Decomposition and <-Decomposition, respectively.

Lemma 5.3.7 *The inference rule* Inductive Rewriting *is sound and safe.*

5.4 Concluding Remarks

We conclude this chapter with a few important remarks.

Firstly, let us summarize the soundness and safeness properties of the inference rules of our calculus for inductive proofs in the following theorem.

Theorem 5.4.1 *All the inference rules presented in this chapter (and thus in this thesis) are sound and safe.*[12]

[12]provided the underlying admissible specifications have *free* constructors

Secondly, we would like to justify our point of view that the requirements with regard to the inference rules, which were dealt with in Section 2.2.3, are met by the presented calculus for inductive proofs to a large extent.

- Obviously, the calculus is goal-directed; and to some degree at least, the form of its inference rules resembles that of a sequent calculus for first-order predicate logic (for more details, refer to Wirth & Kühler, 1995, Section 1.3) Furthermore, our applicative inference rules are based on a concept of explicitly applicable induction hypotheses and lemmas. All in all, the calculus appears to be reasonably comprehensible[13] and hence *user-oriented*.
- The calculus is also *expressive* in that it provides (i) inference rules such as e.g. Substitution Addition and Literal Addition for (user-guided) case analyses and the (ii) inference rules Non-Inductive Subsumption and Inductive Subsumption for (user-guided) explicit instantiations of lemmas and induction hypotheses, respectively. Moreover, all of the inference rules are roughly modeled on what we consider elementary proof steps. For instance, applications of inference rules such as Constant Rewriting, Non-Inductive Rewriting or Inductive Rewriting yield single rewrite steps.
- The basic suitability of the calculus to (partially) *automated* proof control is grounded on the fact that formulas in goals are restricted to first-order clauses made up of equational, definedness or order literals. Furthermore, we are able to present a comprehensive inductive proof strategy in Chapter 8 that is capable of constructing proofs of (simple) inductive theorems with the proposed inference rules *without* any user-guidance.
- Finally, observe that all the inference rules of the calculus are safe, a property which is sufficient to guarantee the *refutational soundness* of the presented framework for inductive theorem proving (see Theorem 6.2.5).

[13]According to our experience, new users to QUODLIBET usually require little time for getting sufficiently familiar with the inference rules needed in "standard" proof constructions.

Chapter 6

Proof State Graphs

After presenting the specification language, the induction order and the calculus for inductive proofs in the preceding chapters, we now introduce the concept of a proof state graph as the last main component of our formal framework for inductive theorem proving. Proof state graphs strongly resemble AND/OR graphs, which were developed for general problem solving strategies. Essentially, they explicitly record the proof dependencies arising from applications of inference rules in proof attempts. On the basis of the proposed notion of a proof state graph, we can finally make precise what constitutes a *proof* (graph) in our framework and state the main soundness result of this thesis: Every clause that occurs in a proof (graph) constructed with the (sound) inference rules of our calculus for inductive proofs is inductively valid with respect to the given specification. Recall that requirements related to proof state graphs as a means of representing proof constructions were discussed in Section 2.2.4.

This chapter is organized as follows. We give a formal definition and examples of proof state graphs in 6.1. Furthermore, we identify (partial) proof attempts as certain subgraphs of proof state graphs and define proof graphs. Thereafter (in 6.2), we present the two soundness results of Part I of this thesis: Our framework for inductive theorem proving is sound as well as refutationally sound. We conclude the chapter (in 6.3) with a few assessing remarks.

6.1 Representing Proof Constructions

In the foregoing chapter we introduced the inference rules of our calculus for inductive proofs, each of which can be used to reduce a goal $\langle \Gamma ; w \rangle$ to a sequence of (sub-) goals. Moreover, we showed that for the reduction of $\langle \Gamma ; w \rangle$ an *applicative* inference rule may make use of a further goal $\langle \Pi ; \hat{w} \rangle$ by applying $\langle \Pi ; \hat{w} \rangle$ to $\langle \Gamma ; w \rangle$ either non-inductively or as an induction hypothesis. However, we have not made precise so far what properties a sequence of applications of inference rules must have in order to yield a *proof* of an inductive theorem. In this section we give an answer to this question by explaining how proof state graphs enable us to represent the proof dependencies that determine the states of proof attempts. These proof dependencies among the goals in proof attempts arise from reductions of goals to subgoals and from non-inductive or inductive applications of goals to other goals.

Basically, proof state graphs are directed labeled graphs on subsets of the natural numbers, and therefore it seems appropriate to briefly compile the elementary notions and notations from graph theory that are used throughout the thesis.[1] In what follows, a *graph* $G = (N, E, L)$ consists of a non-empty finite set $N \subseteq \mathbb{N}$ of vertices or *nodes*, a set $E \subseteq N \times N$ of directed edges or *arcs* and a *labeling function* L that maps the nodes and arcs to elements of certain sets (see below). Let $G = (N, E, L)$ be such a graph and $\nu, \nu' \in N$. We say that, if $(\nu, \nu') \in E$ then ν' is a *successor* of ν. A node without successors (in G) is called a *leaf* (in G), whereas all other nodes are called *internal nodes*. A graph $G' = (N', E', L')$ is called a *subgraph* of G if $N' \subseteq N$, $E' \subseteq E$ and $L' = L|_{N' \cup E'}$. A (directed) *path* from ν to ν' is a finite sequence $\nu_0, \ldots, \nu_n$ of nodes in N such that $\nu_0 = \nu$, $\nu_n = \nu'$, $(\nu_{i-1}, \nu_i) \in E$ for $i = 1, \ldots, n$ and none of the nodes in $\nu_0, \ldots, \nu_n$ occurs more than once.

Lemma 6.1.1 *Suppose $\nu_0, \ldots, \nu_n$ and $\mu_0, \ldots, \mu_m$ are two paths in a graph G such that $\nu_n = \mu_0$ and $\nu_0 \neq \mu_m$. Then there is a path from ν_0 to μ_m in G.*

A sequence $\nu_0, \ldots, \nu_n, \nu_0$ of nodes in N is a (directed) *cycle* if $\nu_0, \ldots, \nu_n$ is a path with $n > 0$ and $(\nu_n, \nu_0) \in E$. Moreover, G is called *cyclic* if it contains a (directed) cycle and *acyclic* otherwise. Recall that an undirected tree is a connected and acyclic undirected graph. Now G is a (directed) *tree* if the undirected graph associated with G is an undirected tree. Furthermore, a directed tree G is called *rooted* if there is a unique node ν in G (the root) such that $(\nu', \nu) \notin E$ for each $\nu' \in N$.

For every proof state graph $G = (N, E, L)$, we have $N = N_{\mathrm{axiom}} \uplus N_{\mathrm{goal}} \uplus N_{\mathrm{inf}}$. That is, every node in a proof state graph is either (i) an *axiom node* labeled with a goal $\langle \Pi\,;\, () \rangle$ that consists of the clause representation Π of a defining rule and the empty tuple as weight, (ii) a *goal node* labeled with any goal or (iii) an *inference node*, which is labeled with information describing a particular inference step. A proof state graph initially consists of axiom nodes for the defining rules of the given specification (see Def. 6.1.2(a)) and of single goal nodes, one for each of the conjectures to be proved (Def. 6.1.2(b)), and it grows by "expanding" goal nodes. In every *expansion step* (Def. 6.1.2(c)), an inference rule is applied to the goal labeling the node to be expanded, and new goal nodes are created for the subgoals resulting from the application of the inference rule. Besides, a (dummy) inference node is generated in every expansion step that may e.g. describe the corresponding inference step.

For the ensuing definition of a proof state graph, we still need to explain how we partition any subset of $\mathbb{N}$ into three subsets for axiom, goal and inference nodes. Given any $M \subseteq \mathbb{N}$ we uniformly associate three subsets with M as follows:

(1) $M_{\mathrm{axiom}} = \{\, n \in M \mid n \equiv 0 \bmod 3 \,\}$ (2) $M_{\mathrm{goal}} = \{\, n \in M \mid n \equiv 1 \bmod 3 \,\}$
(3) $M_{\mathrm{inf}} = \{\, n \in M \mid n \equiv 2 \bmod 3 \,\}$

[1] Nevertheless, we assume familiarity with the basic concepts of graph theory.

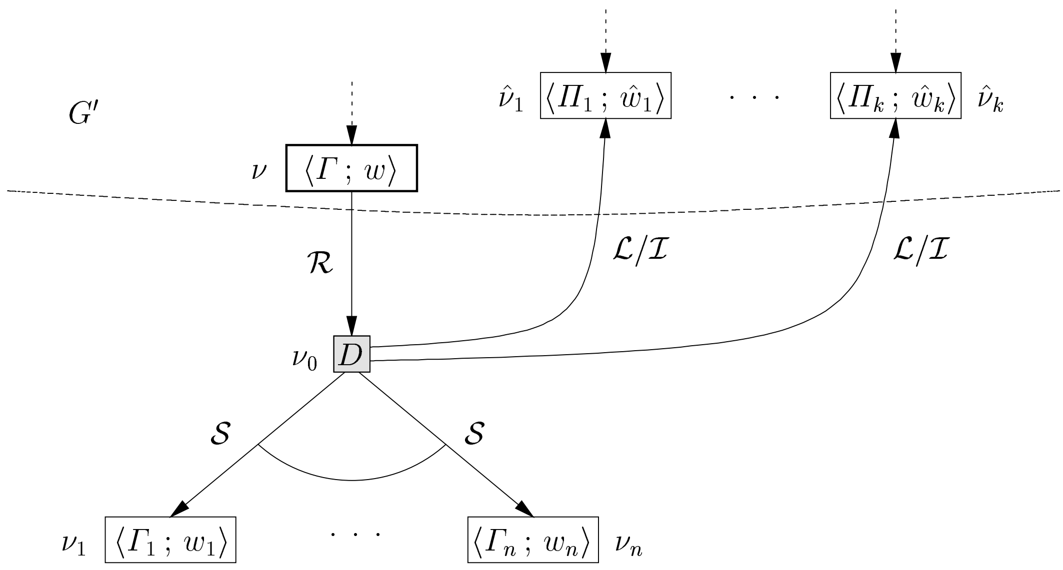


Figure 6.1: General expansion step for a goal node ν in G'

Definition 6.1.2 Let *spec* $=$ (sig, C, R) be an admissible specification. The set of *proof state graphs* w.r.t. *spec* is inductively defined by:

(a) Let $\Pi_1, \ldots, \Pi_m$ be the clause representations of some of the defining rules in R, and let $\nu_1, \ldots, \nu_m \in \mathbb{N}_{\text{axiom}}$. Then $G = (\{\nu_1, \ldots, \nu_m\}, \emptyset, L)$ is a proof state graph w.r.t. *spec* if $L = \{(\nu_i, \langle\Pi_i\,;\,()\rangle\,) \mid i \in \{1, \ldots, m\}\}$.

(b) Let $G' = (N', E', L')$ be a proof state graph w.r.t. *spec*, let $\langle\Gamma\,;\,w\rangle$ be a goal (over *sig* and V), and let $\nu \in \mathbb{N}_{\text{goal}} \backslash N'_{\text{goal}}$. Then $G = (N' \cup \{\nu\}, E', L' \cup \{(\nu, \langle\Gamma\,;\,w\rangle)\})$ is a proof state graph w.r.t. *spec*.

(c) Let $G' = (N', E', L')$ be a proof state graph w.r.t. *spec* and let $\nu \in N'_{\text{goal}}$ such that $L'(\nu) = \langle\Gamma\,;\,w\rangle$. Assume that for $n, k \in \mathbb{N}$ there are goals $\langle\Gamma_1\,;\,w_1\rangle, \ldots, \langle\Gamma_n\,;\,w_n\rangle$ and $\langle\Pi_1\,;\,\hat w_1\rangle, \ldots, \langle\Pi_k\,;\,\hat w_k\rangle$ such that

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

is an instance of an inference rule,[2] where $U_1, \ldots, U_k \in \{\mathcal{L}, \mathcal{I}\}$.

Let $\nu_0 \in \mathbb{N}_{\text{inf}} \backslash N'_{\text{inf}}$, and let $\nu_1, \ldots, \nu_n \in \mathbb{N}_{\text{goal}} \backslash N'_{\text{goal}}$. Furthermore, suppose that there are $\hat\nu_1, \ldots, \hat\nu_k \in N'_{\text{goal}} \cup N'_{\text{axiom}}$ such that, for $j = 1, \ldots, k$, $L'(\hat\nu_j) = \langle\Pi_j\,;\,\hat w_j\rangle$ and $U_j = \mathcal{I}$ implies $\hat\nu_j \in N'_{\text{goal}}$. Then $G = (N, E, L)$ is a proof state graph w.r.t. *spec* provided that (see Figure 6.1)

- $N = N' \cup \{\nu_0, \ldots, \nu_n\}$
- $E = E' \cup \{(\nu, \nu_0)\} \cup \{(\nu_0, \nu_i) \mid i \in \{1, \ldots, n\}\} \cup \{(\nu_0, \hat\nu_j) \mid j \in \{1, \ldots, k\}\}$

[2]as defined in Notation 4.1.3

– $L = L' \cup \{((\nu,\nu_0),\mathcal{R})\} \cup \{(\nu_0, D)\} \cup \{(\nu_i, \langle \Gamma_i\,;\, w_i\rangle) \mid i \in \{1,\dots,n\}\}$
$\cup\, \{((\nu_0,\nu_i),\mathcal{S}) \mid i \in \{1,\dots,n\}\} \cup\, \{((\nu_0,\hat{\nu}_j),U_j) \mid j \in \{1,\dots,k\}\}$[3]
where D may be a (possibly non-existent) description of the given instance of the inference rule.

An example of a proof state graph with respect to a specification with constructors is given in Figure 6.2. The proof state graph represents a proof — by *mutual* induction — of the total definedness of the two mutually recursive operators even and odd. Note that we usually omit the $\mathcal{R}$- and $\mathcal{S}$-labels in graphical presentations of proof state graphs. Figure 2.1 in Section 2.3 depicts another example of a proof state graph w.r.t. the specification $spec_{\mathsf{plus}}$, for which a more "economical" presentation mode was chosen.

With regard to the definition of proof state graphs, a few remarks should be made. Let $G = (N, E, L)$ be a proof state graph. First of all, any axiom node $\nu \in N_{\text{axiom}}$ must be a leaf in G, as only goal nodes can be expanded. Besides, every arc leading to an axiom node is $\mathcal{L}$-labeled. In other words, if $\nu \in N_{\text{axiom}}$ then for every $\nu' \in N$ with $(\nu',\nu) \in E$, we have $\nu' \in N_{\text{inf}}$ and $L((\nu',\nu)) = \mathcal{L}$. Secondly, G is a *bipartite* graph: Every arc $(\nu,\nu') \in E$ has the property that either $\nu \in N_{\text{goal}}$ and $\nu' \in N_{\text{inf}}$ or $\nu \in N_{\text{inf}}$ and $\nu' \in (N_{\text{goal}} \cup N_{\text{axiom}})$. Thirdly, G may be *cyclic*, i.e. contain directed cycles (see Figures 2.1 and 6.2). However, the subgraph G' obtained from G by removing all the arcs labeled with $\mathcal{L}$ or $\mathcal{I}$ in G is *acyclic*,[4] and the components of G' are rooted (directed) trees.

Definition 6.1.3 Suppose $G = (N, E, L)$ is a proof state graph w.r.t. an admissible specification. Let $G' = (N, E', L')$, where $E' = \{(\nu,\nu') \in E \mid L((\nu,\nu')) \in \{\mathcal{R},\mathcal{S}\}\}$ and $L' = L|_{N \cup E'}$.

(1) G' is said to be the *proof state forest* corresponding to G.

(2) If the root of a component of G' is a goal node (and hence not an axiom node), then we call that component a *proof state tree* in G.

For instance, there is one proof state tree in the proof state graph of Figure 2.1, while the proof state graph in Figure 6.2 contains two proof state trees, the roots of which are labeled with the goals $\langle \mathrm{def}(\mathsf{even}(x))\,;\, x\rangle$ and $\langle \mathrm{def}(\mathsf{odd}(x))\,;\, x\rangle$, respectively. Note that the roots of the proof state trees in a proof state graph G are exactly those goal nodes that were introduced according to case (b) of Definition 6.1.2 in the construction of G.

It was said in Section 2.2.4 that by including proof state graphs in our framework for inductive theorem proving we seek to be capable of supporting *multiple* proof attempts for the same conjecture. As a result of this requirement we admit so-called choice points in proof state graphs. A goal node $\nu \in N_{\text{goal}}$ is said to be a *choice point* in a proof

[3] We call an arc labeled with one of $\mathcal{R},\mathcal{S},\mathcal{L}$ or $\mathcal{I}$ a *r*eduction, a *s*ubgoal, a *l*emma or an *i*nduction hypothesis arc, respectively.

[4] even the undirected graph associated with G' does not have any undirected cycles

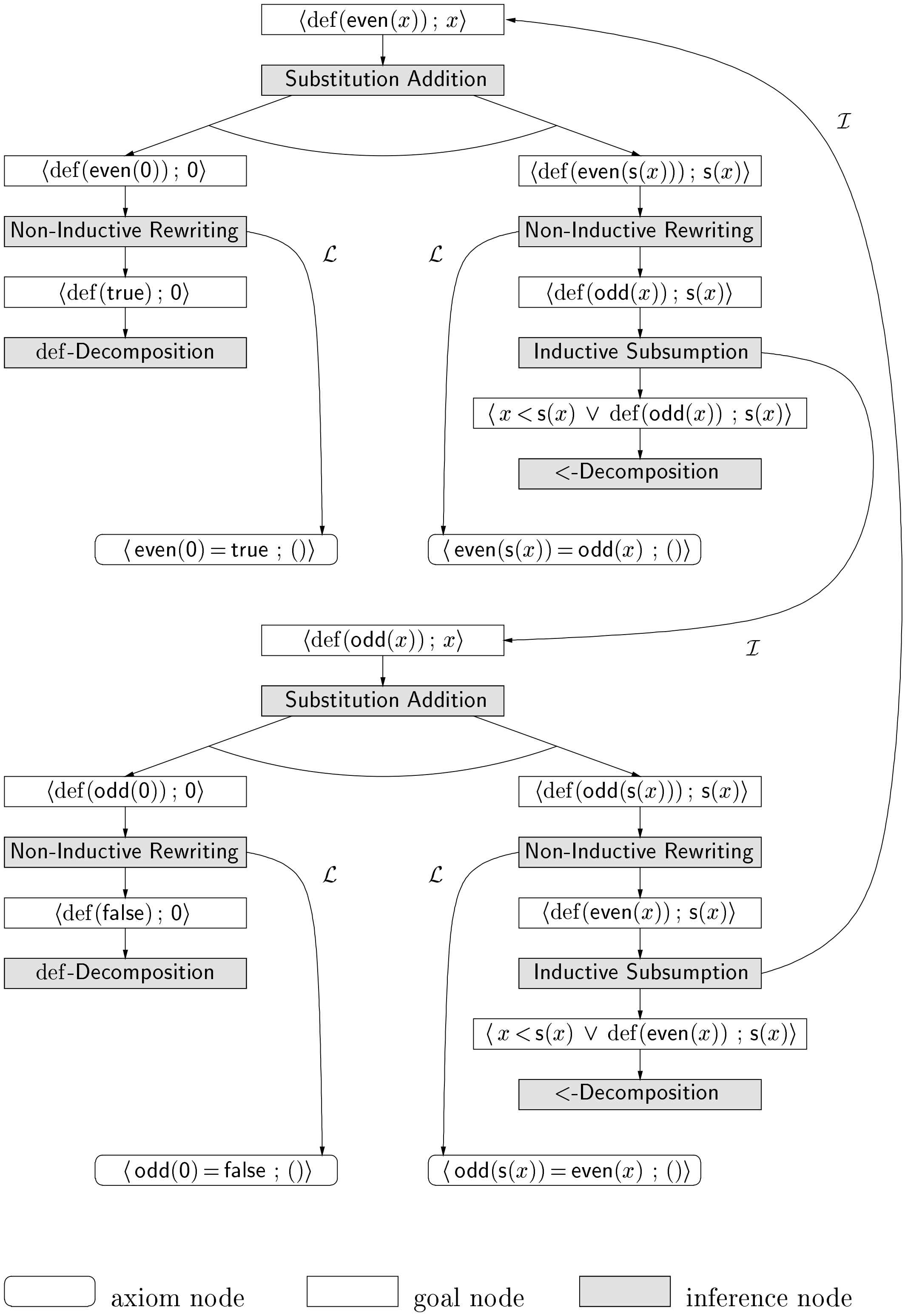


Figure 6.2: A proof (state) graph

state graph G if ν has more than one successor in G (which must be inference nodes). This means that the user is enabled to expand goal nodes *more than once*, and hence it is possible to represent different applications of inference rules to the same goal within one proof state graph. A schematic example of a proof state graph with a choice point is given in Figure 6.3. The goal node ν_3 is a choice point as it has two successors.

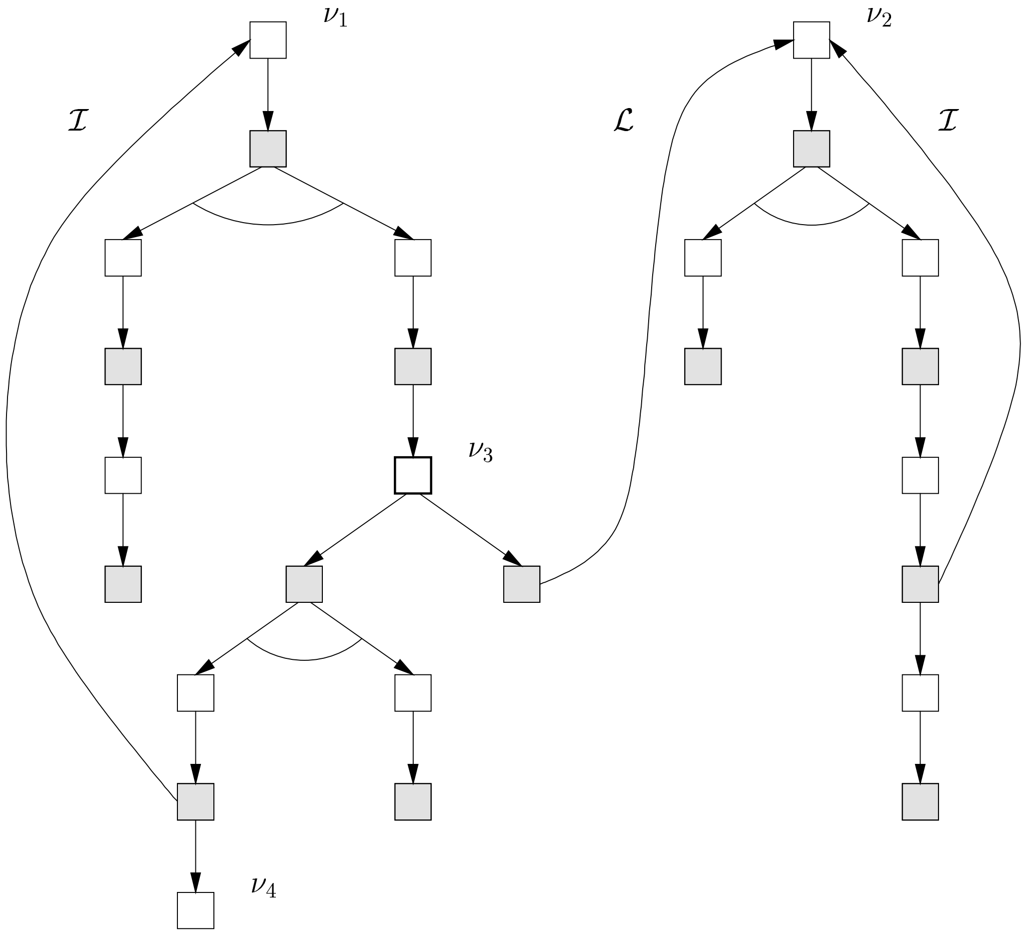


Figure 6.3: Schematic example of a proof state graph with a choice point

An obvious resemblance between proof state graphs on the one hand and *AND/OR graphs*, which are used in general problem solving strategies (see e.g. Pearl, 1984), on the other hand, is illustrated in Figure 6.3. Goal nodes can be considered to correspond to OR nodes whereas inference nodes assume the role of AND nodes: (1) In order to reduce the problem of proving a goal $\langle \Gamma\,;\, w\rangle$ represented by a goal node ν to problems given by subgoals, only *one* application of an inference rule to $\langle \Gamma\,;\, w\rangle$ has to be found. Hence, only *one* of the successors of (a choice point) ν is needed for a proof of $\langle \Gamma\,;\, w\rangle$. (2) As *each* subgoal arising from the application of an inference rule to $\langle \Gamma\,;\, w\rangle$ must be proved to yield a proof for $\langle \Gamma\,;\, w\rangle$, *all* the successors of an inference node have to be considered.

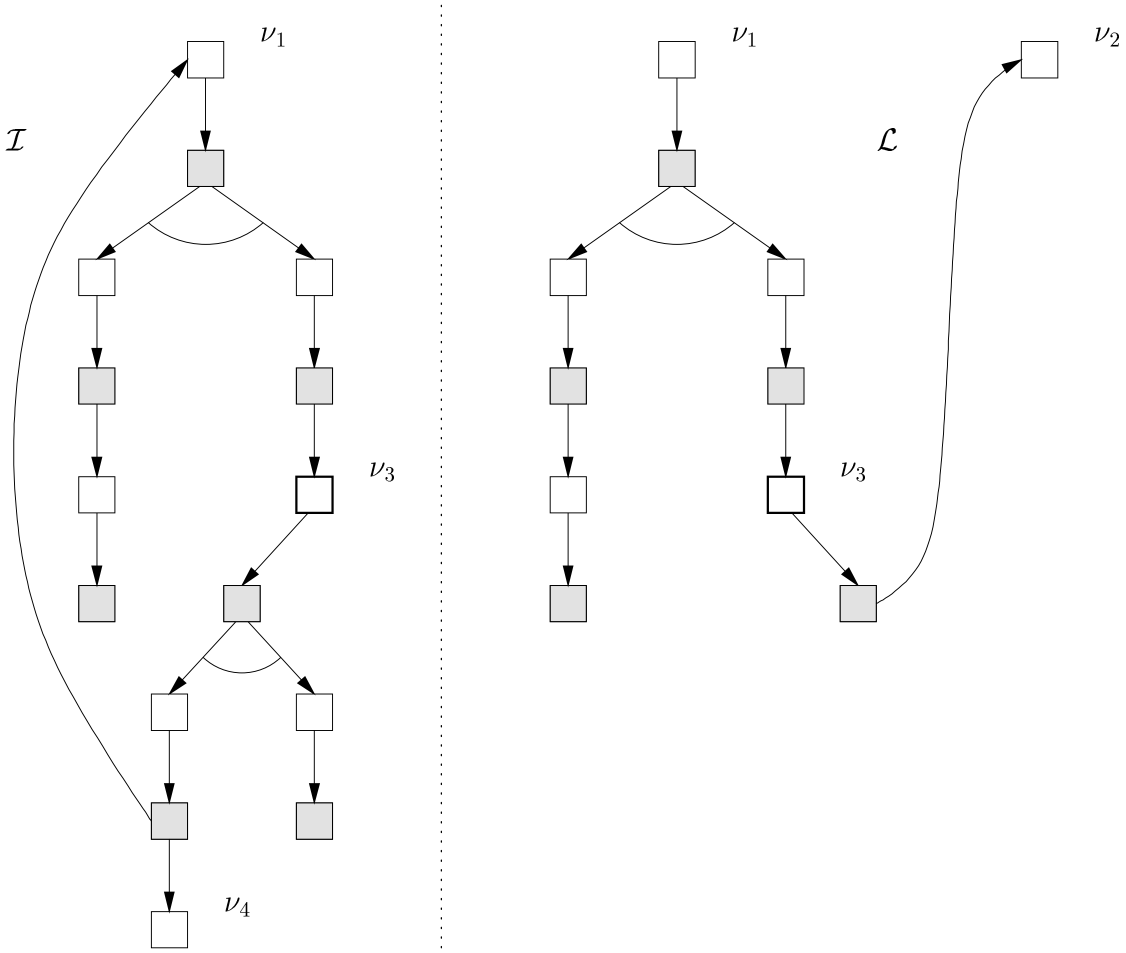


Figure 6.4: The proof attempts for ν_1

Intuitively seen, a choice point gives rise to different proof attempts in the same proof state graph. For example, the proof state graph depicted in Figure 6.3 represents two proof attempts for (the goal labeling) ν_1 that differ in the way the choice point ν_3 is expanded (see Figure 6.4). The proof attempt on the left-hand side still contains an "open" goal node (ν_4), but the right one is actually a "proof graph" for ν_1, since there is also a "proof graph" for the "lemma node" ν_2 in the proof state graph of Figure 6.3. We are now going to give formal definitions of proof attempts, open goal nodes, lemma nodes and proof graphs. As one can see below, proof attempts will turn out to be *partial* proof attempts that are maximal in the given proof state graph. We provide the "constructive" (i.e. inductively defined) notion of a partial proof attempt in order to be able to formulate a suitable invariant for the construction of proof state graphs (Lemma 6.2.2), which will be applied in the proof of our major soundness result in Section 6.2.1.

A path $\nu_0, \ldots, \nu_n$ is called *$\mathcal{L}$-free* if $L((\nu_{i-1}, \nu_i)) \neq \mathcal{L}$ for $i = 1, \ldots, n$. We define partial proof attempts as follows.

Definition 6.1.4 Suppose $G = (N, E, L)$ is a proof state graph w.r.t. *spec*. The set of *partial proof attempts* for goal nodes in G is inductively defined by:

(a) For $\tilde{\nu} \in N_{\text{goal}}$, $P = (\{\tilde{\nu}\}, \emptyset, L|_{\{\tilde{\nu}\}})$ is a partial proof attempt for $\tilde{\nu}$ in G.

(b) Let $P' = (N', E', L')$ be a partial proof attempt for some $\tilde{\nu} \in N_{\text{goal}}$ in G, and let $\nu' \in N'_{\text{goal}}$ such that (1) ν' is a leaf in P' and (2) there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν' in P'. Suppose ν' was *expanded* in G, i.e. there are $\nu_0, \nu_1, \ldots, \nu_n, \hat{\nu}_1, \ldots, \hat{\nu}_k \in N$ such that $(\nu', \nu_0) \in E$ and $\nu_1, \ldots, \nu_n, \hat{\nu}_1, \ldots, \hat{\nu}_k$ are exactly the successors of ν_0 in G where $L((\nu_0, \nu_i)) = \mathcal{S}$ for $i = 1, \ldots, n$ and $L((\nu_0, \hat{\nu}_j)) \in \{\mathcal{L}, \mathcal{I}\}$ for $j = 1, \ldots, k$. Then $P = (\tilde{N}, \tilde{E}, \tilde{L})$ is a partial proof attempt for $\tilde{\nu}$ in G where

- $\tilde{N} = N' \cup \{\nu_0, \nu_1, \ldots, \nu_n, \hat{\nu}_1, \ldots, \hat{\nu}_k\}$
- $\tilde{E} = E' \cup \{(\nu', \nu_0), (\nu_0, \nu_1), \ldots, (\nu_0, \nu_n), (\nu_0, \hat{\nu}_1), \ldots, (\nu_0, \hat{\nu}_k)\}$
- $\tilde{L} = L|_{\tilde{N} \cup \tilde{E}}$.

We call $\tilde{\nu}$ the *start node* of P. Two (of the numerous) partial proof attempts for ν_3 in the proof state graph of Figure 6.3 are depicted in Figure 6.5. Observe that the partial proof attempt on the right-hand side cannot grow any further, because the path from ν_3 to its only leaf ν_2 is not $\mathcal{L}$-free.

It is easily verified that every partial proof attempt for a goal node in G is a connected subgraph of G and a proof state graph w.r.t. *spec*. Moreover, as each goal node that is expanded in the construction of a partial proof attempt must be a leaf, there can be no choice-points in partial proof attempts. Furthermore, a simple structural induction shows that if P is a partial proof attempt for a goal node $\tilde{\nu}$ in G, then there is a path from $\tilde{\nu}$ to ν in P for each node ν in P.

The following lemma characterizes those goal nodes in a partial proof attempt P that cannot be reached through an $\mathcal{L}$-free path beginning in the start node of P. Recall that a partial proof attempt cannot "grow" beyond such nodes (see Definition 6.1.4).

Lemma 6.1.5 *Let $P = (N, E, L)$ be a partial proof attempt for a goal node $\tilde{\nu}$ in a proof state graph G. Suppose $\nu \in N_{\text{goal}}$ such that there is no $\mathcal{L}$-free path from $\tilde{\nu}$ to ν in P. Then ν is a leaf in P and $L((\nu', \nu)) = \mathcal{L}$ for every $\nu' \in N_{\text{inf}}$ with $(\nu', \nu) \in E$.*

We call such nodes lemma nodes of the partial proof attempt P, while the other leaves in P are said to be open goal nodes of P:

Definition 6.1.6 Let $P = (N, E, L)$ be a partial proof attempt for a goal node $\tilde{\nu}$ in a proof state graph G.

(i) $\nu \in N_{\text{goal}}$ is called an *open goal node* of P if ν is a leaf in P and there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν in P.

(ii) A goal or an axiom node $\nu \in N$ is called a *lemma node* of P if there is a $\nu' \in N_{\text{inf}}$ such that $(\nu', \nu) \in E$ and $L((\nu', \nu)) = \mathcal{L}$.

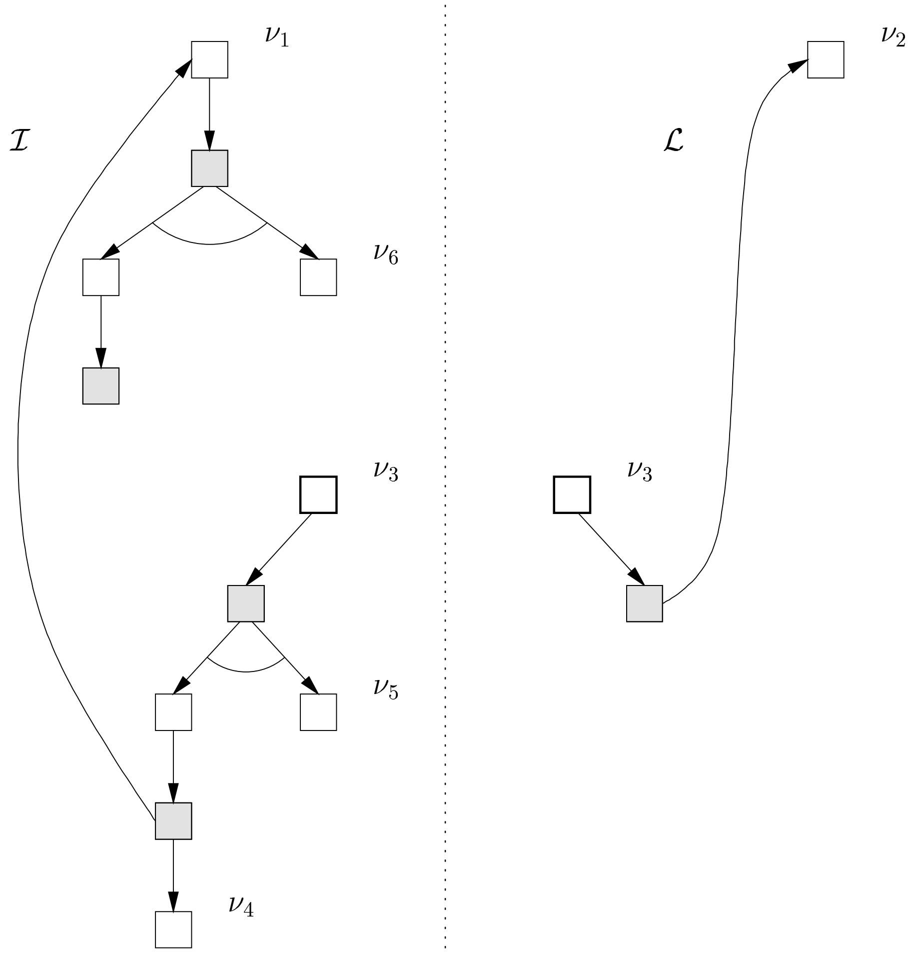


Figure 6.5: Partial proof attempts for ν_3

Note that a goal node ν can be both an open goal node and a lemma node of a partial proof attempt P. In this case there must be two paths from the start node of P to ν one of which is $\mathcal{L}$-free. Moreover, it is evident that the lemma nodes of P do not depend on the start node of P — they can be determined without knowing the start node of P. That the same is true for the open goal nodes of P is stated in the following lemma.

Lemma 6.1.7 *For $i \in \{1, 2\}$, let P_i be a partial proof attempt for a goal node $\tilde{\nu}_i$ in G. Suppose ν is a goal node in G and $P_1 = P_2$. Then ν is an open goal node of P_1 iff ν is an open goal node of P_2.*

We use $\mathrm{OGNd}(P)$ for the open goal nodes and $\mathrm{LNd}(P)$ for the lemma nodes of P.

We can finally define proof attempts and proof graphs in a proof state graph. A proof attempt in a proof state graph G is a partial proof attempt that is *maximal* in that each of its open goal nodes is a leaf in G. Moreover, a proof graph in G is a proof attempt that does not have any open goal nodes (i.e. all its leaves are inference nodes or lemma nodes) and whose lemma nodes are labeled with goals that are *known* to be inductively valid. Recall that a proof graph is to represent a (complete) proof of the inductive validity of the goal labeling its start node.

Definition 6.1.8 Suppose $G = (N, E, L)$ is a proof state graph w.r.t. *spec* and $\tilde{\nu}$ is a goal node in G.

(i) A partial proof attempt P for $\tilde{\nu}$ in G is said to be a *proof attempt* for $\tilde{\nu}$ in G if for each $\nu \in \mathrm{OGNd}(P)$, ν is a leaf in G.

(ii) We call a proof attempt P for $\tilde{\nu}$ in G a *proof graph* for $\tilde{\nu}$ in G if $\mathrm{OGNd}(P) = \emptyset$ and if for each $\nu \in \mathrm{LNd}(P)$ with $L(\nu) = \langle \Pi\,;\, \hat{w} \rangle$, Π is inductively valid w.r.t. *spec*.

For example, the proof state graph depicted in Figure 6.3 contains two proof attempts for the goal node ν_1 (see Figure 6.4), the second of which is also a proof graph for ν_1.[5] Moreover, both proof state graphs in Figures 2.1 and 6.2 are actually proof graphs for the roots of the proof state trees they contain. Finally, observe the obvious similarity between proof graphs in proof state graphs and solution graphs in AND/OR graphs. Besides, partial proof attempts directly correspond to the so-called solution bases used in searching AND/OR graphs (see Pearl, 1984).

6.2 Soundness Results

In this section we discuss two soundness results, which may be considered the major theoretical achievements of this thesis: Our framework for inductive theorem proving is sound as well as refutationally sound.

6.2.1 Soundness

Speaking in general terms, a calculus is commonly said to be sound if every formula that has a *syntactic* proof in the calculus is also a *semantic* consequence of the given axioms, i.e. the formula is valid in every model "of interest" of the axiomatization. Having made precise (i) what constitutes a proof (graph) and (ii) what an inductive theorem w.r.t. an admissible specification is (see Definition 3.4.3), we can now formulate what soundness means in the case of our framework for inductive theorem proving. For the framework to be called *sound*, we require that the clause in the goal labeling the start node of any *proof graph* be inductively valid with respect to the given specification (Theorem 6.2.4). In the proof of this soundness result, we make use of a lemma that in some sense can be regarded as a (loop) *invariant* for the construction of proof state graphs (Lemma 6.2.2).

In order to make this invariant sufficiently powerful for a proof of Theorem 6.2.4 we first have to distinguish certain so-called *inductive* open goal nodes of a partial proof

[5] There is a proof graph for (the lemma node) ν_2 in the proof state graph of Figure 6.3.

attempt. We call a path $\nu_0, \ldots, \nu_n$ *$\mathcal{R}/\mathcal{S}$-labeled* if $L((\nu_{i-1}, \nu_i)) \in \{\mathcal{R}, \mathcal{S}\}$ for each $i = 1, \ldots, n$.

Definition 6.2.1 Let $P = (N, E, L)$ be a partial proof attempt for a goal node $\tilde{\nu}$ in a proof state graph G. A goal node $\nu \in P$ is said to be an *inductive* open goal node of $\tilde{\nu}$ and P if $\nu \in \mathrm{OGNd}(P)$ and there is no $\mathcal{R}/\mathcal{S}$-labeled path from $\tilde{\nu}$ to ν in P.

By $\mathrm{IOGNd}(\tilde{\nu}, P)$, we denote the set of inductive open goal nodes of $\tilde{\nu}$ and P. For example, let P_1 be the partial proof attempt for ν_3 on the left-hand side in Figure 6.5. We have $\mathrm{OGNd}(P_1) = \{\nu_4, \nu_5, \nu_6\}$, but $\mathrm{IOGNd}(\nu_3, P_1) = \{\nu_6\}$, since there are $\mathcal{R}/\mathcal{S}$-labeled paths from ν_3 to ν_4 and from ν_3 to ν_5 in P_1. We name such open goal nodes "inductive", because every path from the start node of a partial proof attempt to an inductive open goal node must involve an application of an *inductive* applicative inference rule, i.e. contain an arc labeled with $\mathcal{I}$.

Now our invariant for proof state graphs states that the following key property of partial proof attempts with regard to $\mathcal{A}$-counterexamples and the induction order $\precsim_{\mathcal{A}}$ holds after every step in the construction of proof state graphs:[6]

Lemma 6.2.2 (Invariant for the construction of proof state graphs) *Let $\mathcal{A}$ be a data model of an admissible specification with free constructors spec. Suppose that $G = (N, E, L)$ is a proof state graph w.r.t. spec and that $\tilde{\nu} \in N_{\mathrm{goal}}$ with $L(\tilde{\nu}) = \langle \tilde{\Gamma} \,;\, \tilde{w} \rangle$. For any partial proof attempt P for $\tilde{\nu}$ in G, one of the following statements holds:*

(1) *There is a $\nu \in LNd(P)$ such that $L(\nu) = \langle \Pi \,;\, \hat{w} \rangle$ and Π is not inductively valid w.r.t. spec.*

(2) *For any $\mathcal{A}$-counterexample of the form $(\langle \tilde{\Gamma} \,;\, \tilde{w} \rangle, \tilde{\sigma}, \tilde{\varphi})$ there is a $\nu \in OGNd(P)$ with $L(\nu) = \langle \Gamma \,;\, w \rangle$ and an $\mathcal{A}$-counterexample of the form $(\langle \Gamma \,;\, w \rangle, \sigma, \varphi)$ such that*

   (a) $(\langle \Gamma \,;\, w \rangle, \sigma, \varphi) \precsim_{\mathcal{A}} (\langle \tilde{\Gamma} \,;\, \tilde{w} \rangle, \tilde{\sigma}, \tilde{\varphi})$ *and*

   (b) $\nu \in IOGNd(\tilde{\nu}, P)$ *implies* $(\langle \Gamma \,;\, w \rangle, \sigma, \varphi) \prec_{\mathcal{A}} (\langle \tilde{\Gamma} \,;\, \tilde{w} \rangle, \tilde{\sigma}, \tilde{\varphi})$

In the straightforward but extensive proof of Lemma 6.2.2, we assume that every clause occurring in a lemma node of the partial proof attempt P is inductively valid w.r.t. *spec* and show that then (2) is true for P and the $\mathcal{A}$-counterexample $(\langle \tilde{\Gamma} \,;\, \tilde{w} \rangle, \tilde{\sigma}, \tilde{\varphi})$. We do this by structural induction on partial proof attempts in combination with well-founded induction on the set of $\mathcal{A}$-counterexamples with respect to $\precsim_{\mathcal{A}}$. The central "tool" used in this induction proof is the subsequent lemma, which provides us with suitable subgraphs of P to which the induction hypothesis can be applied.

Lemma 6.2.3 *Let $P = (N, E, L)$ be a partial proof attempt for a goal node $\tilde{\nu}$ in a proof state graph G. Suppose $\hat{\nu} \in N_{\mathrm{goal}}$ such that there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to $\hat{\nu}$ in P. Then the following statements hold:*

(1) *There is a unique maximal subgraph $\hat{P}$ of P such that*

[6] For $\mathcal{A}$-counterexamples and the induction order $\precsim_{\mathcal{A}}$ refer to Section 4.1.

(i) $\hat{P}$ *is a partial proof attempt for* $\hat{\nu}$ *in* P

(ii) $OGNd(\hat{P}) \subseteq OGNd(P)$ *and*

(iii) $LNd(\hat{P}) \subseteq LNd(P)$.

(2) *If there is also an* $\mathcal{L}$*-free path from* $\hat{\nu}$ *to* $\tilde{\nu}$ *in* P*, then there is a unique and maximal subgraph* $\hat{P}$ *of* P *such that*

(i) $\hat{P}$ *is a partial proof attempt for* $\hat{\nu}$ *in* P *with* $\tilde{\nu} \in OGNd(\hat{P})$

(ii) $OGNd(\hat{P})\backslash\{\tilde{\nu}\} \subseteq OGNd(P)$ *and*

(iii) $LNd(\hat{P}) \subseteq LNd(P)$.

It may be interesting to observe that each of the two statements in Lemma 6.2.3 is proved by defining an appropriate relation $\longmapsto$ on partial proof attempts, which is terminating and strongly confluent, and by showing that $\hat{P}$ defined to be the (unique) normal form of $(\{\hat{\nu}\}, \emptyset, L|_{\{\hat{\nu}\}})$ w.r.t. $\longmapsto$ has the required properties.

The following theorem guarantees the soundness of the proposed framework for inductive theorem proving (see above). The first claim of the theorem is virtually a corollary to Lemma 6.2.2, whereas Theorem 6.2.4(2) provides a further and stronger result, namely that the clause in a goal labeling *any* goal node in a proof graph is inductively valid with respect to the given specification.

Theorem 6.2.4 (Soundness) *Let spec be an admissible specification with free constructors. Suppose that* $G = (N, E, L)$ *is a proof state graph w.r.t. spec and that* P *is a proof graph for a goal node* $\tilde{\nu} \in N_{\mathrm{goal}}$ *in* G.

(1) *If* $L(\tilde{\nu}) = \langle \tilde{\Gamma}\,;\, \tilde{w}\rangle$ *then* $\tilde{\Gamma}$ *is inductively valid w.r.t. spec.*

(2) *If* ν *is a goal node in* P *and* $L(\nu) = \langle \Gamma\,;\, w\rangle$ *then* Γ *is inductively valid w.r.t. spec.*

6.2.2 Refutational Soundness

By *refutational soundness* of a calculus or — more generally — of a formal framework for (inductive) theorem proving we mean that, informally speaking, one can *correctly* infer the falseness of a conjecture Γ from the derivation of an obviously contradictory formula, such as the empty clause, in a proof attempt for Γ within the framework. For an example, consider the proof attempt for the conjecture $\mathsf{less}(\mathsf{0}, x) = \mathsf{true}$ in Figure 6.6, which contains a goal with the empty clause $\square$. Since our framework for inductive theorem proving is indeed refuationally sound (see below), we are allowed to conclude from the construction of the proof state graph depicted in Figure 6.6 that the clause $\mathsf{less}(\mathsf{0}, x) = \mathsf{true}$ is not an inductive theorem w.r.t. $spec_{\mathsf{div}}$ (see Example 2.2.1).

The refutational soundness of our framework for inductive theorem proving follows from the subsequent theorem.

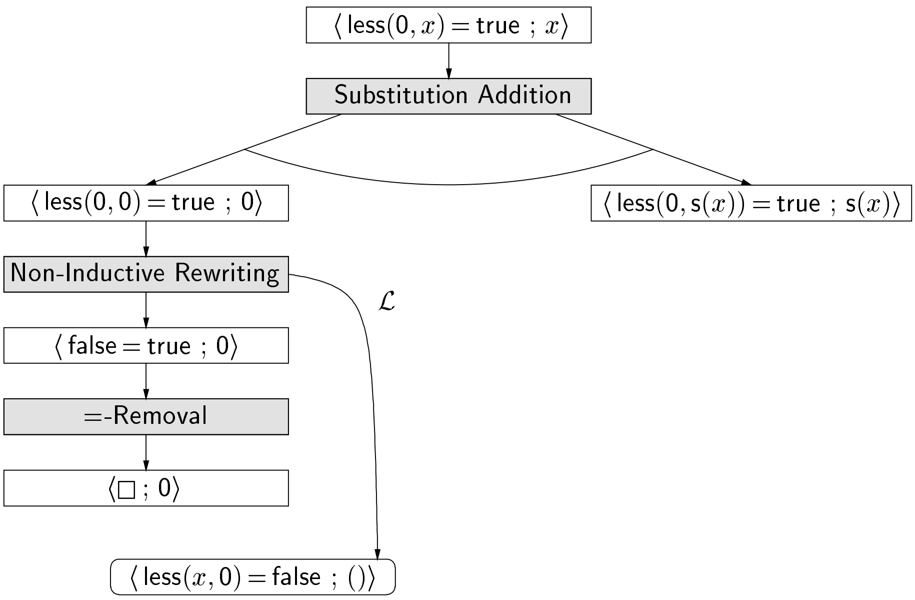


Figure 6.6: A proof attempt for a false conjecture

Theorem 6.2.5 (Refutational Soundness) *Let spec be an admissible specification with free constructors. Assume that $G = (N, E, L)$ is a proof state graph w.r.t. spec and that $\nu_1, \ldots, \nu_n \in N_{\text{goal}}$ are exactly the roots of the proof state trees in G.*[7]

If, for $i = 1, \ldots, n$, $L(\nu_i) = \langle \Gamma_i \,;\, w_i \rangle$ and Γ_i is inductively valid w.r.t. spec then, for any $\nu \in N_{\text{goal}}$ with $L(\nu) = \langle \Gamma \,;\, w \rangle$, Γ is inductively valid w.r.t. spec.

So why does Theorem 6.2.5 entail refutational soundness? Suppose a proof state graph G w.r.t. *spec* contains a goal node ν that is labeled with a goal of the form $\langle \Box \,;\, w \rangle$. Let G' be the component of G to which ν belongs. Clearly, G' is also a proof state graph w.r.t. *spec*. By Theorem 6.2.5 (contraposition), the root of one of the proof state trees in G' must be labeled with a goal of the form $\langle \Gamma' \,;\, w' \rangle$ such that Γ' is *not* inductively valid w.r.t. *spec*. Consequently, we can infer the falseness of at least one of the "input" conjectures from the derivation of a goal with the empty clause.

Evidently, the clause $\mathsf{less}(0, x) = \mathsf{true}$ is not an inductive theorem w.r.t. $spec_{\mathsf{div}}$, as there is only one proof state tree in Figure 6.6. The schematic example of a proof state graph in Figure 6.7 contains two proof state trees, the roots of which are ν_1 and ν_5. Still, we can conclude that if ν_2 is labeled with a goal of the form $\langle \Box \,;\, w \rangle$ then the clause in the goal labeling ν_1 cannot be inductively valid. The reason for this is that we can apply Theorem 6.2.5 to the subgraph which contains ν_1, ν_2 and ν_3 and whose open goal nodes are ν_2 and ν_3. Given that only ν_4 is labeled with a goal of the form $\langle \Box \,;\, w \rangle$, however, Theorem 6.2.5 only allows us to infer that one of the "input" conjectures in the goals labeling ν_1 and ν_5 cannot be an inductive theorem.

[7] see Definition 6.1.3

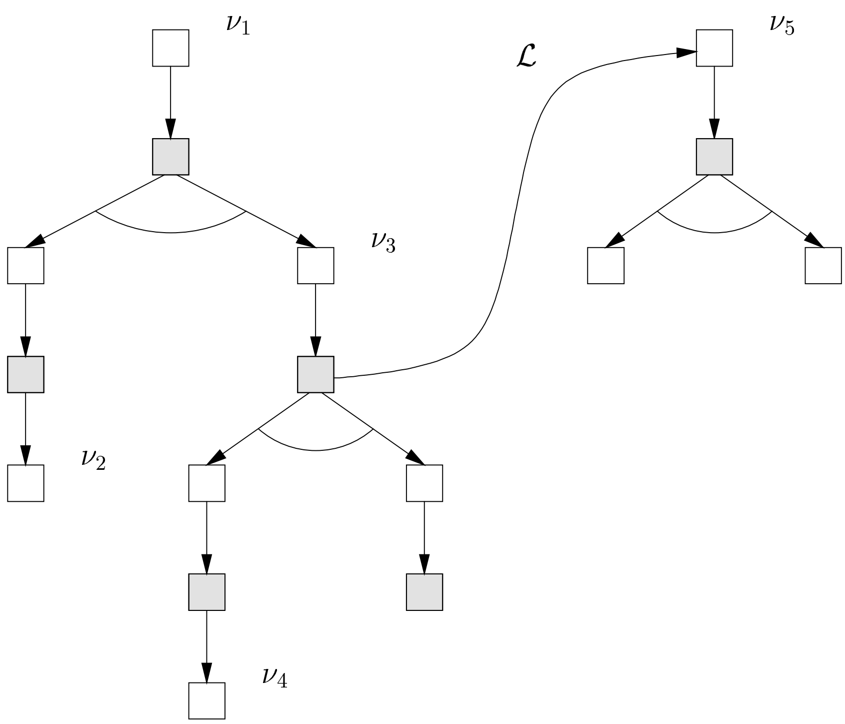


Figure 6.7: Schematic example of a proof state graph

6.3 Concluding Remarks

Let us conclude this chapter with two remarks on (i) the soundness results of the preceding section and on (ii) completeness properties of our framework for inductive theorem proving and with (iii) a discussion of the question as to whether the presented concept of a proof state graph meets the requirements stated in Section 2.2.4.

First of all, it should be noted that neither in the definition of proof state graphs nor in our proofs of the soundness results any use is made of *specific* properties of the concrete inference rules of our calculus for inductive proofs or of the proposed induction order with the corresponding notion of a weight (see Section 4.2). To be more precise, the proofs of Lemma 6.2.2 and Theorem 6.2.4 are solely based on the existence of a well-founded quasi-order $\precsim_{\mathcal{A}}$ on the set of $\mathcal{A}$-counterexamples for every data model $\mathcal{A}$ of the given specification and on the notion of a *sound* inference rule as defined in Notation 4.1.3 and Definition 4.1.4, whereas in the proof of Theorem 6.2.5 we only need the abstract properties of a *safe* inference rule (see Notation 4.1.3 and Definition 5.1.1). This means in particular that we can *easily* extend or modify our calculus for inductive proofs without endangering the achieved soundness results: We just have to make sure that every (new) inference rule matches the general format of Notation 4.1.3 and is sound as well as safe.

Secondly, as a consequence of Gödel's *First Incompleteness Theorem*, our framework for inductive theorem proving cannot be *complete* in the sense that, for every inductive theorem Γ w.r.t. an admissible specification with free constructors *spec*, there is a proof state graph w.r.t. *spec* which contains a proof graph for a node labeled with a goal of the form $\langle \Gamma\,;\, w \rangle$. Moreover, our framework for inductive theorem proving is not even *refutationally complete* (see Bachmair, 1988). Although the equation $\mathsf{div}(0,0) = 0$ is not inductively valid w.r.t. $spec_{\mathsf{div}}$ from Example 2.2.1, we cannot *refute* it within our framework. By this we mean that it is impossible to construct a proof state graph w.r.t. $spec_{\mathsf{div}}$ which contains a node labeled with a goal of the form $\langle \Box\,;\, w \rangle$ and to which Theorem 6.2.5 can be applied in order to establish that $\mathsf{div}(0,0) = 0$ is not inductively valid w.r.t. $spec_{\mathsf{div}}$.[8]

Thirdly, we provide evidence for our claim that the proposed concept of a proof state graph fulfills the requirements with regard to the representation of proof constructions that are listed in Section 2.2.4: The example of the proof state graph in Figure 6.2 illustrates the fact that our framework for inductive theorem proving supports the *lazy* generation of induction hypotheses and *mutual* induction. Moreover, Definition 6.1.2 entails that applications of *unproved* lemmas are possible, and the discussion of choice points in Section 6.1 made it clear that *multiple* complete or incomplete proof attempts for the same conjecture may occur in proof state graphs. Finally, the notion of a proof graph (see Definition 6.1.8) and Theorem 6.2.4 yield the required criterion for recognizing inductive validity that is expressed in terms of graph theoretical properties of proof attempts. As a consequence, many clauses occurring in proof constructions can be identified as inductively valid long before the proof construction has been completed. Note that the problem of determining proof graphs in proof state graphs can be solved by adapting well-known methods of searching solution graphs in AND/OR graphs (see Pearl, 1984) to the situation given by proof state graphs.

[8]Due to the underlying notion of inductive validity (see Definition 3.4.3), which is based on validity in a *non-trivial* model class, achieving refutational completeness appears to be extremely difficult if not impossible.

Part II

The Inductive Theorem Prover QuodLibet

Chapter 7

An Introduction to QuodLibet

In the foregoing Part I of this thesis, we presented an elaborate formal framework for inductive theorem proving, which comprises (i) an equational specification language, (ii) an induction order, (iii) a calculus for inductive proofs and (iv) a concept of a proof state graph for the representation of proof constructions. Due to its soundness properties, its comprehensiveness and user-orientation, the proposed framework is well suited to constitute the *precisely defined* and *comprehensible* logical or theoretical foundations of a practically adequate inductive theorem prover.

Mainly in order (1) to demonstrate the practical adequacy of this formal framework in "real-life" problem domains, but also (2) to develop (and evaluate) a new concept for flexible user-guided proof control with adaptable degree of automation, the author of this thesis has designed and implemented[1] an inductive theorem prover that is based on the formal framework of Part I as its logical foundations. The prover was named QuodLibet, as will be explained below. From the fact that the logical foundations of the prover meet the requirements discussed in Section 2.2, which was shown in Chapters 3 – 6, we can immediately derive the following of QuodLibet's essential properties:

- QuodLibet's specification language facilitates adequate formalizations of data types with partial operations; it admits positive/negative conditional equations as axioms and has easily verifiable admissibility conditions.
- QuodLibet provides a user-oriented and expressive calculus for inductive proofs, which consists of explicitly defined — and thus identifiable — inference rules and which is based on a practically adequate semantic induction order. Besides, the prover supports the explicit application of goals as induction hypotheses.
- By making use of proof state graphs as a means of explicitly representing and managing the proof dependencies arising in proof constructions, QuodLibet is capable of supporting the lazy generation of induction hypotheses, mutual induction, applications of unproved lemmas and multiple complete or incomplete proof attempts for the same conjecture.

[1]with the assistance of graduate students Christian Embacher (1995), Robert Eschbach, Tobias Schmidt-Samoa (1997), Jürgen Schumacher and Christof Sprenger (1996)

The software system QuodLibet is the overall subject of the second major part of this thesis. In this and the subsequent chapter of Part II, we present QuodLibet by discussing a relevant selection of its interesting aspects. In particular, we are going to (1) list some of the general objectives that we laid down for the development of the prover, (2) briefly deal with the software architecture of QuodLibet, (3) describe the functionality of the system and (4) sketch its graphical user interface. The focus of Part II, however, is on (5) a new approach to more flexible forms of (user-guided) proof control with adaptable degree of automation (see Chapter 8). This approach is based on so-called (proof) *tactics*, for which we developed a special proof control language named QML. QuodLibet allows the user (i) to apply the tactics made available by the prover — these tactics range from tactics for trivial simplification steps to tactics representing comprehensive inductive proof strategies — and (ii) to write new tactics in QML, which may be integrated into the system to extend its functionality. In order to enable the reader to assess the proposed concept for flexible user-guided proof control, we give a few interesting examples of tactic-based proof constructions in Appendix E.

This first chapter of Part II is organized as follows. In 7.1 we begin our introduction to QuodLibet by describing general requirements for an inductive theorem prover that is to be based on the formal framework discussed in the first part of this thesis. After that (in 7.2), we briefly explain the high level system structure of the software system QuodLibet. Section 7.3 then deals with the functionality of the prover in terms of its command language, while essential properties of QuodLibet's graphical user interface are presented in 7.4.

7.1 Requirements for QuodLibet

Virtually as a lead-in to QuodLibet, let us first discuss some general requirements that we hold to be *practically* important for an inductive theorem prover based on the formal framework of Part I, before we are going to describe concrete properties of QuodLibet in the subsequent sections.

A first and rather natural requirement follows from the fact that the formal framework presented in Part I is to constitute the logical foundations of the prover: We demand that, speaking in general terms, the essential achievements of the formal framework, which can be found in Section 2.2, find expression in corresponding features of the prover. To be more precise, (1) we expect the prover to "know" all the inference rules of the proposed calculus for inductive proofs "by name" and make them available to the user; and (2) we expect the prover to provide the proof state graphs of the formal framework, which comprise proof state trees (see Definition 6.1.3), as *system objects*. In particular, the prover should allow the user

- to *navigate* through proof state trees so that he can expand any goal node at any time,
- to expand goal nodes more than once, which leads to *choice points* in proof state graphs, and

- to analyze proof state graphs by means of adequate *display* facilities.

Closely related to these demands is a further, fundamental requirement, which is also due to the great emphasis we place on user interaction (see Section 2.2.3): The prover has to be capable of carrying out — in cooperation with the user if necessary — *any* proof construction that is feasible within the (more abstract) formal framework of Part I.

Another essential requirement states that the prover has to *rigorously* check all steps to be taken in axiomatizations of data types or in proof constructions and *reject* them if necessary. In particular, (1) we require the prover to guarantee that each admissibility condition of the proposed specification language (see Definition 3.2.4) holds for any specification accepted by the prover. Note that because of the confluence criterion in Theorem 3.3.2, this requirement can be met relatively easily. Moreover, (2) the prover has to ensure that for every application of an inference rule in a proof construction, the corresponding applicability conditions are provably fulfilled. This means for instance that a method of testing whether or not a given set of constructor substitutions constitutes a cover set of substitutions (see Definition 5.2.14) needs to be implemented for the inference rule Substitution Addition (see Figure 5.9). Observe in this context that there are inductive theorem provers, such as e.g. UNICOM (Gramlich & Lindner, 1991), which accept potentially inadmissible specifications. As a consequence, the correctness of proof constructions carried out with a prover of this kind is relative to the validity of certain admissibility assumptions.[2]

Rather practical yet important is the requirement of a *graphical user interface* (GUI). Due to the role proof state graphs play in the proposed formal framework for inductive theorem proving, a prover with a user interface capable of graphical presentations — and manipulations — of proof state graphs can significantly enhance the user-friendliness of the prover by supporting the visualization of proof constructions. A GUI should also increase the usability of the prover, i.e. the effort needed to learn and operate the prover properly. Nevertheless, especially experienced users may appreciate another kind of user interface, namely a (common) teletype style interface that provides the functionality of the prover in form of a more efficiently usable *command interpreter*. A further advantage of such a simpler interface could be the increased portability of the prover, i.e. a reduction of the effort required to transfer the prover from one hardware or operating system configuration to another.

Last but not least we put special emphasis on practically adequate forms of *proof control*. According to our definition, the proof control of an inductive theorem prover lays down how, speaking in general terms, the construction (or search) of proofs by the prover and/or the user is organized. We distinguish two basic forms of proof control, namely user-guided or *interactive* proof control on the one hand and programmed or *automated* proof control on the other hand. Various inductive theorem provers, however, have hybrid forms of proof control. For example, the Boyer-Moore prover

[2] such as "The rewrite relation associated with the specification is assumed to be terminating."

NQTHM is an inductive theorem prover with a complex and elaborate proof procedure forming its essentially automated proof control (see Boyer & Moore, 1979), but there are a few parameters allowing the user to *globally* influence the way the prover searches for proofs — to a limited extent though.[3] Another example of an inductive theorem prover with a hybrid form of proof control is LP, the LARCH PROVER (Garland & Guttag, 1989, 1991). Although its proof control is essentially interactive, LP does not require the user to determine every single application of the available (elementary) inference rules: LP provides a small and fixed set of user commands for automatically solving *simple* proof tasks, such as e.g. normalizing terms in conjectures with certain "harmless" (i.e. terminating and unconditional) equations and rewrite rules.

In view of the user's *active* role required for successfully employing an inductive theorem prover (see Section 2.2.3), we demand a concept of proof control that offers a great(er) deal of *flexibility* as to the forms of proof control it supports. Without going too much into detail, let us state more precisely what we require in particular: The prover is to provide an extensive and easily *extensible* set of user commands that allow the user

- to direct the prover to carry out any possible application of an inference rule from Chapter 5 to any goal node in the current proof state graph, thereby facilitating completely interactive proof control on a *low level*
- to invoke comprehensive inductive proof strategies in order to obtain complete proofs for simple conjectures constructed by the prover, which realizes (partially) automated proof control on a *high level* and
- to delegate various (isolated) proof tasks on *intermediate levels* to the prover such as generating induction schemes, doing case analyses based on the definitions of recursive operations, simplifying goals with axioms or lemmas etc.

Observe that we are also interested in enabling the (competent) user (i) to modify the intermediate or higher level "inference operators" supplied by the prover or (ii) to add new ones — given that the effort required for this can be kept within acceptable limits. As a consequence, we should avoid the disadvantages of "hard-wired" (automated) proof control and be able to utilize the prover as a tool for the development of new heuristics and (sub-) strategies for the automation of inductive theorem proving.

The requirements discussed in this section can roughly be summarized as follows: We intend to develop an inductive theorem prover that is to serve as a user-friendly, flexible and reasonably "intelligent" *proof assistant* to the user. This is partly expressed in the name that we have chosen for our prover: The Latin words "quod libet" mean *as you wish* or, more freely translated, *whatever you* (i.e. the user) *like*.[4] It is our objective that QUODLIBET can effectively assist the user in what we consider the *proof engineering* process required for proofs of non-trivial "real-life" inductive theorems.

[3]These parameters are the so-called "hints" of Boyer and Moore (1988). We speak of proof *global* influence on the proof control — or *indirect* interaction — in the case of NQTHM, because its proof procedure does not have any *interaction points* at which the user is prompted to interact with the prover.

[4]*was beliebt* in German

7.2 The System Structure of QuodLibet

We continue our introduction to QuodLibet by sketching the system structure of our inductive theorem prover. Figure 7.1 contains a strongly simplified representation of the software architecture of XQuodLibet[5] showing the top-level decomposition of the system into its main components (or subsystems) *Inference Machine*, *User Interfaces* and *Proof Control Unit*.

By software architecture, we mean the output of the *system design* phase in the process of developing a software system (see e.g. Jalote, 1991). Roughly stated, the software architecture of a software system describes the complete (recursive) decomposition of the system into subsystems down to the level of modules, and it defines the interfaces of the subsystems and modules making up the software system. The system design of QuodLibet was documented by Kühler et al. (1998) using a slightly extended version of *MIL*, the *M*odule *I*nterconnection *L*anguage (introduced by DeRemer & Kron, 1976), a both graphical and textual architecture description language (see Embacher, 1995). Currently, XQuodLibet comprises about one hundred modules, roughly three fourths of which were implemented in Common LISP, while the remaining modules were written in Tcl/Tk and belong to the graphical user interface. In spite of our experience that MIL has appeared to be a really useful architecture description language during the development of XQuodLibet, we consider the less technical and presumably more intuitive representation of XQuodLibet's high level system structure in Figure 7.1 to be more appropriate in this context than a corresponding description in MIL.

The *inference machine* forms the kernel of QuodLibet. The resources provided at the interface of this major subsystem essentially comprise — besides the basic abstract data types required for syntactic objects such as e.g. sorts, terms, substitutions, weights, clauses or proof state trees — a set of so-called *inference machine operations*. This set includes operations for

- initializing the three data units of the inference machine (see Figure 7.1), which determine the state of the inference machine (see below)
- defining sorts together with their constructors and declaring other function symbols (the "defined operators")
- introducing and deleting defining rules (i.e. axioms)
- generating proof state trees that consist of single goal nodes for conjectures
- *single* applications of the inference rules presented in Chapter 5 and
- deleting complete proof state trees and subtrees of proof state trees that root in inference nodes.

The three data units mentioned above make up the memory of the inference machine in that each of them stores the currently existing objects of the kind indicated by the

[5] Sometimes we use the name XQuodLibet in order to emphasize the fact that QuodLibet has an X-based graphical user interface.

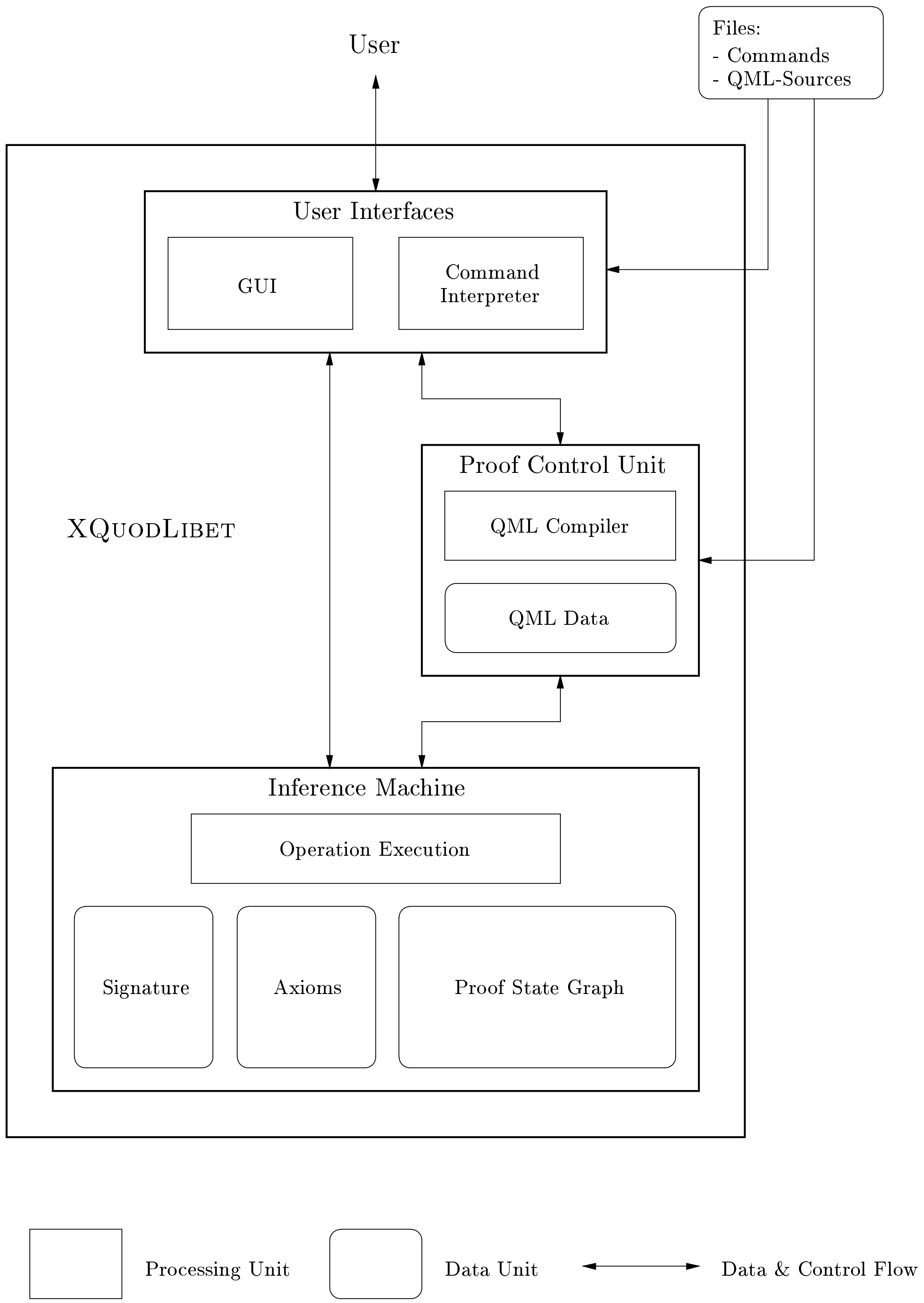


Figure 7.1: High level system structure of XQUODLIBET

name of the data unit.[6] Therefore, they represent the current state of the inference machine.

Note that, due to the observed principle of *information hiding* in the system design, the internal structure of the inference machine is hidden to the other subsystems User Interfaces and Proof Control Unit. That is, only by generating appropriate calls of the provided inference machine operations can these subsystems effect any activities of the inference machine, which may be required e.g. for the axiomatization of data types or the construction of inductive proofs.

The subsystem *User Interfaces* realizes the graphical user interface of XQuodLibet, into which a command interpreter is integrated. Note that the prover allows the user complete access to its functionality not only through the windows, menus and buttons of the GUI, but also through the use of the command interpreter. Furthermore, the command interpreter is capable of executing sequences of user commands that are read in from command files. We will introduce the command language in Section 7.3, while the GUI is the subject of Section 7.4.

For the purpose of low level interactive proof control, XQuodLibet offers the user certain (graphical as well as textual) user commands most of which the subsystem User Interfaces directly translates into calls of the inference machine operations listed above. For (partially) automated proof control on intermediate and higher levels, however, the user has to make use of the subsystem *Proof Control Unit* (PCU). Roughly speaking, the PCU realizes the *programmed control of the inference machine*: It may be employed (1) to compile tactics, i.e. routines written in a special proof control language called QML for controlling the inference machine, and (2) to load the generated code into the data unit *QML Data*, thus dynamically extending the functionality of the system. Moreover, (3) the PCU is needed to execute the code of previously compiled tactics, that may be invoked by the user through a certain user command or called within other tactics. As will be shown in Section 8.3, a set of useful QML routines is provided as part of XQuodLibet. These routines currently range from tactics for trivial simplification steps to tactics representing comprehensive inductive proof strategies.

7.3 The Command Language of QuodLibet

Up to this point the introduction to our inductive theorem prover in this chapter has been rather general. In the two foregoing sections, we discussed our main requirements for the development of QuodLibet and sketched its high level system structure. In this section we present the *functionality of the system* in more concrete terms. To this end we could describe either the windows, menus and buttons of the graphical user interface of XQuodLibet or the command language of the command interpreter,

[6]Technically seen, these three data units actually correspond to four *abstract data object modules* in the software architecture of the inference machine: one for the current signature, one for the current set of defining rules and two for the current proof state graph.

as each of the functions provided by the system at its user interface can be invoked through the GUI as well as the command interpreter. Mainly for reasons of clarity and comprehensibility we are going to use the command language of the command interpreter to describe the functionality of QUODLIBET.

Note that in this section, we do not treat the command language systematically but roughly explain, mostly based on simple examples, how the user can work with the system in order to formalize data types (in 7.3.1) and *interactively* construct inductive proofs (in 7.3.2). Further miscellaneous commands are the subject of 7.3.3, whereas the commands needed to utilize the proof control unit for more flexible forms of proof control will be introduced in Chapter 8. Moreover, a small example of a complete proof construction is given in 7.3.4. For a complete definition of the syntax of the command language in EBNF we refer to Appendix B. How the command interpreter, which is integrated into XQUODLIBET's GUI, can be used to operate the system will be dealt with in Section 7.4.

7.3.1 Commands for Formalizing Data Types

Roughly speaking, the state of QUODLIBET is given by the contents of the data units in the inference machine and the proof control unit (see Figure 7.1). All these data units are empty, when the system is in its initial state. In particular, there are neither predefined signature objects, such as sorts, constructors or other function symbols, nor any defining rules in the system after its start, as opposed to e.g. NQTHM (Boyer & Moore, 1988). As a consequence, the user is always required to formalize the data type[7] of interest with QUODLIBET, before he can use the system to construct any proofs. To formalize a data type D, the user has to determine a specification with constructors $spec = (sig, C, R)$ that is admissible and whose inductive semantics $\mathrm{DMod}(spec)$ corresponds to D.[8]

The prover expects the user to define the sorts S of the signature $sig = (S, F, \alpha)$ and the constructors C at first. For this purpose, QUODLIBET provides the `define sort` command which introduces a single sort $s \in S$ *together* with all the constructors of result sort s. For instance, by entering

```
define sort Bool with constructors
   true, false : --> Bool .
```

the user defines a sort called `Bool` with two constant constructors for the truth values, whereas the following two commands

```
define sort Nat with constructors
   0 : --> Nat
   s : Nat --> Nat .
```

[7] A precise characterization of data types is given in the beginning of Section 3.1.
[8] see Definitions 3.2.4 and 3.4.3, respectively

```
define sort ListNat with constructors
   nil  :             --> ListNat
   cons : Nat ListNat --> ListNat .
```

define sorts `Nat` for the natural numbers and `ListNat` for lists of natural numbers with the commonly used constructors, respectively.

Each argument sort s_i of a constructor in the definition of a sort s must be either equal to s or known to the system due to a preceding sort definition or *sort declaration* for s_i. For example, the `declare sorts` command is actually needed in the following definitions of the two *mutually dependent* sorts `s1` and `s2`:

```
declare sorts s2 .

define sort s1 with constructors
  c : --> s1
  g : s2 --> s1 .

define sort s2 with constructors
  d : --> s2
  f : s1 --> s2 .
```

Observe that QuodLibet urges the user to define sort `s2` (i) after executing the `declare sorts` command for `s2` and (ii) after executing the `define sort` command for `s1`, because at these points, there do not exist any constructor ground terms of sort `s2` — the current signature is not *sensible* then (see Section 3.1.1). Also note that other inductive theorem provers, such as e.g. Inka or Nqthm, do not allow the user to introduce mutually dependent sorts.[9]

Variables occurring in defining rules or goals must have been declared prior to their first use. This may be done with the `declare variables` command, which also associates a sort with every declared variable. Moreover, the system needs to know of what kind every declared variable is: Recall from Section 3.1.1 that in our framework for inductive theorem proving, a variable is either a *constructor* variable or a *general* variable. The two example commands below declare some constructor variables, which are practically more relevant than general variables, and a general variable for the sorts defined in foregoing examples.

```
declare constructor variables
  b : Bool
  x, y, z : Nat
  l, l1, l2, l3 : ListNat .

declare general variables B : Bool .
```

[9] The so-called *shell* principle of Nqthm is even more restrictive (see Boyer & Moore, 1988, p. 18).

Having defined the sorts corresponding to the data domains of the given data type and declared sufficiently many variables, the user can now introduce and axiomatize the (non-constructor) operations of the data type with the `define operator` command. The purpose of this command is to specify the arity of a new operator and describe its "behavior" (with regard to data items represented by constructor ground terms) in terms of defining rules (see Definition 3.2.1). For instance, given sort definitions and variable declarations as above, the usual (irreflexive) order relation on the natural numbers, the constant function 1, the subtraction as well as the division can be axiomatized with QUODLIBET as follows (see $spec_{\mathsf{div}}$ in Example 2.2.1):

```
define operator  less : Nat Nat --> Bool  with defining rules
less-1:
  less(0,s(y))    = true
less-2:
  less(x,0)       = false
less-3:
  less(s(x),s(y)) = less(x,y) .

define operator  1 : --> Nat  with defining rules
one-1:
  1 = s(0) .

define operator  minus : Nat Nat --> Nat  with defining rules
minus-1:
  minus(x,0)       = x
minus-2:
  minus(s(x),s(y)) = minus(x,y) .

define operator  div : Nat Nat --> Nat  with defining rules
div-1:
  div(x,y) = 0
  if
    y =/= 0,
    less(x,y) = true
div-2:
  div(x,y) = s(div(minus(x,y),y))
  if
    y =/= 0,
    less(x,y) =/= true,
    def less(x,y) .
```

Let us make a few remarks on the `define operator` command and these examples. Firstly, QUODLIBET expects a unique name for every defining rule entered by the user. The name of the first defining rule for the "predicate" `less` e.g. is `less-1`. Secondly, given a defining rule of the form $l = r \leftarrow \varDelta$ in a `define operator` command for a new operator f, the command language requires that $\mathrm{top}(l) = f$ and l be linear. Thirdly, the system only accepts a `define operator` command, if it succeeds in proving that the specification resulting from extending the current specification[10] with the defining rules occurring in this command remains admissible (see Definition 3.2.4). Note that the confluence test which is needed for a verification of the admissibility conditions by QuodLibet is based on computing critical pairs and checking their complementarity as suggested by Theorem 3.3.2. Finally, observe that executing the three example commands yields an admissible specification, even though the defining rules `minus-1`, `minus-2`, `div-1` and `div-2` specify the subtraction and division on the natural numbers only *incompletely* — which is appropriate considering the fact that *partial* operations are to be formalized. The reader is asked to refer to Appendix B for the concrete syntax of terms, literals, defining rules, clauses etc.

The command language provides a second command for introducing defining rules, namely the `assert` command. This command allows the user to add *any* defining rules to the current set of defining rules stored in the inference machine on condition that the system is capable of proving that the intended extension also yields an admissible specification. The `assert` command may be useful when "incomplete" axiomatizations of previously defined operations have to be extended. For instance, the commonly used total extensions of the subtraction operation `minus` and the division operation `div`, as axiomatized in the example above, can be obtained by entering

```
assert
minus-3:
  minus(0,y) = 0
div-3:
  div(x,y) = 0
  if
    y = 0 .
```

The command interpreter requires that operators occurring in a `define operator` or `assert` command be known to the system in the sense that they have been defined or *declared* before.[11] Since such "forward" declarations of operators are indeed needed for (natural) axiomatizations of *mutually recursive* operations,[12] the command language includes a `declare operators` command. Consider e.g. the defining rules for `even` and `odd` that label the axiom nodes of the proof state graph in Figure 6.2. The following commands can be used to axiomatize these operations with the system correspondingly.

```
declare operators  odd : Nat --> Bool .

define operator  even : Nat --> Bool  with defining rules
even-1:
  even(0)    = true
even-2:
  even(s(x)) = odd(x) .
```

[10] i.e. the contents of the data units *Signature* and *Axioms* (see Figure 7.1)

[11] except for the operator f in a `define operator` command for f

[12] due to the fact that the parser of the command interpreter parses only one command at a time

```
define operator  odd : Nat --> Bool  with defining rules
odd-1:
  odd(0)    = false
odd-2:
  odd(s(x)) = even(x) .
```

Observe that NQTHM and some other inductive theorem provers do not admit *natural* axiomatizations of mutually recursive operations (see Boyer & Moore, 1979).

We conclude this subsection with a few general remarks on the state of the inference machine, which is given by the contents of its data units (see Figure 7.1). Note that the commands presented in this subsection have an effect just on the data units Signature and Axioms where the current signature and the current specification are stored. First of all, the current signature is always sensible unless there are sorts that have been declared but not defined yet (see above). In this case, the system does not execute any commands that would change the state of the inference machine except for `define sort` and `declare sorts` commands. Secondly, given that the current signature is sensible, the current specification is always admissible. Thirdly, apart from the `declare sorts` command, the execution of every command discussed in this subsection yields a *constructor-consistent* extension of the current specification (see Definition 3.4.1). Hence Theorem 3.4.2, which states the monotonicity of the inductive semantics w.r.t. constructor-consistent extensions, entails one of the crucial properties of QUODLIBET: Inductive theorems w.r.t. the current specification remain inductively valid w.r.t. the specification contained in any succeeding state of the inference machine — provided that none of the state transitions is caused by the execution of a `delete defining rule` command (see Section 7.3.3).

7.3.2 Commands for Constructing Proof State Trees

Once the user has formalized (parts of) the data type of interest with QUODLIBET by defining/declaring (some of) the needed sorts, constructors, variables and operators, he can begin to prove inductive theorems that formalize "true" statements about the (operations of the) given data type. Recall from Part I that inductive theorem proving in our formal framework essentially amounts to constructing proof state graphs which contain proof graphs for goal nodes labeled with the inductive theorems to be proved.[13] The *view* of the currently constructed proof state graph (see Definition 6.1.2) that QUODLIBET presents to the user comprises the collection of *proof state trees* belonging to the current proof state graph (see Definition 6.1.3), along with the defining rules of the current specification that label its axiom nodes. Essentially, the commands dealt with in this subsection enable the user (i) to generate new proof state trees for the conjectures to be proved and (ii) to expand goal nodes in proof state trees by applications of the inference rules introduced in Chapter 5.

[13]If necessary refer to Section 2.3 for an informal and intuitive impression of how one can construct proofs of inductive theorems within the framework proposed in Part I.

A conjecture, i.e. a clause whose inductive validity w.r.t. the current specification is to be proved, can be introduced by the user with the `prove` command. In executing this command for a clause Γ, the prover creates a new proof state tree that consists of a single goal node labeled with the goal $\langle \Gamma\,;\, w \rangle$, where w is a system generated *weight variable* for the new proof state tree. For example, suppose the current specification consists of the above signature objects and defining rules for the sorts `Bool` and `Nat` as well as the operators `less`, `minus` and `div` (not including the defining rules `minus-3` and `div-3`). Now given that the "domain lemma" for the division operation

$$\mathrm{def}(\mathsf{div}(x,y)) \;\vee\; y = \mathsf{0}$$

(see Example 4.3.1) is to be proved with QuodLibet, the command

```
prove { def div(x,y), y = 0 } div-def
```

can be used to add a suitable proof state tree to the current proof state graph, where the parameter `div-def` represents the unique name of the proof state tree required by the system for references to this conjecture. The output of the command interpreter after executing this command is

```
Creating the PS-tree div-def

The current PS-tree is div-def

The current G-node in div-def is

G-node root  "div-def"
 { def div(x,y),
   y = 0 } ;
 w_div-def(x,y)
** current **
```

This output illustrates some further explanations with regard to the `prove` command and proof state trees: First of all, during the execution of a `prove` command, the prover assigns the name of the just created proof state tree to the system variable `current PS-tree` for the *current proof state tree* in the current proof state graph. Secondly, for each generated proof state tree, a system variable `current G-node` for the *current goal node* exists. Thirdly, the weight in the goal that labels the (only) goal node at root position ε (`root`) in the new proof state tree `div-def` is the system generated *weight variable* `w_div-def(x,y)`, which depends on the constructor variables occurring in the conjecture (`x` and `y` in the example).

The `assume` command is the second command provided by the command language that allows the user to create a proof state tree consisting of one goal node for a conjecture. It differs in its effect from the `prove` command only in that the value of the system variable for the current proof state tree is *not* changed as in the case of the `prove` command (see above). The `assume` command should be preferred to the corresponding `prove` command whenever the user intends to apply the conjecture introduced with the command as a lemma, but wants to postpone the proof of the lemma in order to focus on the construction of the current proof state tree. The use of `assume` is illustrated in the example of a construction of a proof state tree in Section 7.3.4.

Once created, a proof state tree grows by expansions of its goal nodes. In every expansion step, one of the inference rules from Chapter 5 is applied to the goal that labels the node to be expanded, and new goal nodes are created for the subgoals resulting from the application of the inference rule as well as an inference node (see Figure 6.1). An expansion of *any* goal node in *any* proof state tree of the current proof state graph can be brought about by the user through the `apply` command. Besides the name of an inference rule, the command interpreter expects certain (actual) *parameters* in an `apply` command that characterize the intended application of the inference rule unambiguously. We refer to Appendix C for a complete description of the twenty-seven inference rules made available by QUODLIBET, which of course includes the (formal) parameters laid down for each inference rule. For instance, suppose we want to expand the root of the proof state tree `div-def` introduced in the previous example by applying the inference rule Literal Addition with the literal $\mathsf{less}(x, y) = \mathsf{true}$ (see Figure 5.10). This can be achieved with the command

```
apply lit-add less(x,y) = true.
```

assuming `div-def` is the current proof state tree and `root` the current goal node in `div-def`. The execution of this command yields the following output by the command interpreter:

```
Applying lit-add to root in div-def
results in two new subgoals

G-node [1^2]
 { less(x,y) =/= true,
   def div(x,y),
   y = 0 } ;
 w_div-def(x,y)
** current **

G-node [1:2]
 { less(x,y) = true,
   def div(x,y),
   y = 0 } ;
 w_div-def(x,y)

The current G-node in div-def is [1^2]
```

With this system output the subsequent points are to be illustrated. First of all, the names of the inference rules that are used in the command language often differ from the names introduced in Chapter 5 (e.g. `lit-add` vs. Literal Addition). Both names for each inference rule can be found in Appendix C. Secondly, goal nodes (and also inference nodes) in proof state trees are referred to by their *positions* in the proof state trees. Note how positions are denoted in the command language. For example, the position 1.2 is represented by `[1:2]`, while `[1:1:1:1:2:1]` or `[1^4:2:1]` can be used for the position 1.1.1.1.2.1. Thirdly, provided that the goal node expanded with the `apply` command was the current goal node, QUODLIBET determines a new current goal node for the concerned proof state tree. The tree traversal method used by the prover for this purpose depends on the system variable `search-strategy` that exists for every proof state tree and can be changed by the user with the `set search-strategy` command. The possible values for `search strategy` are `depth-first` (default), `breadth-first` and `none`.[14]

Before treating the next command, we have to mention two essential design decisions that were made during the development of QUODLIBET and entail certain restrictions of the generality of the proposed concept of a proof state graph. (1) After every expansion of a goal node by the inference machine, which is always the result of an application of an inference rule (and e.g. due to the execution of an `apply` command), the inference machine checks whether there exist (new) *proof graphs* in the current proof state graph for conjectures not proved before. However, this test for inductive validity w.r.t. the current specification, which is based on Theorem 6.2.4, is carried out *only* for "conjectures", i.e. for clauses in goals labeling the roots of the currently existing proof state trees. (2) As opposed to Definition 6.1.2(c), where it is stated that applicative inference rules may make use of *any* goals as lemmas or induction hypotheses that label axiom or goal nodes, QUODLIBET requires that goals to be applied as lemmas or induction hypotheses label axiom nodes or the *roots* of proof state trees in the current proof state graph. To be more precise, the applicative inference rules provided by the prover are

- `axiom-subs` and `axiom-rewrite` for subsumption and rewrite steps with defining rules (i.e. "axioms")
- `lemma-subs` and `lemma-rewrite` for non-inductive subsumption and rewrite steps with the goals of the *roots* of proof state trees (i.e. "lemmas")
- `ind-subs` and `ind-rewrite` for inductive subsumption and rewrite steps with the goals of the *roots* of proof state trees (i.e. "induction hypotheses") and
- `appl-lit-removal` for non-inductively removing redundant literals with defining rules or the goals of the *roots* of proof state trees.[15]

Practical experience of inductive theorem proving, however, shows that certain proof constructions could not be carried out with QUODLIBET in a natural way, if the system did not enable the user to overcome the restrictions just mentioned above for singular goal nodes by means of the `assign-name` command. For this command allows the user

[14] If the value of `search-strategy` is `none` the value of `current G-node` is not changed.

[15] see Appendix C

to assign a name to any goal node in a proof state tree so that the goal of this goal node can be applied as a lemma or as an induction hypothesis. For example, suppose the operator definition for `less` from Section 7.3.1 is extended with the following defining rules

```
not-1:  not(false) = true
not-2:  not(true)  = false        leq-1:  leq(x,y) = not(less(y,x))
```

in order to axiomatize the usual $\leq$-relation on the natural numbers. Consider the following (incomplete) proof state tree `leq-def` for the conjecture $\mathrm{def}(\mathsf{leq}(x,y))$

```
+-G-node root  "leq-def"
  |  { def leq(x,y) } ;
  |  w_leq-def(x,y)
  |
  +-I-node [1]
    | < axiom-rewrite 1 [1] leq-1 1 [ ] >
    |
    +-G-node [1^2]
      |  { def not(less(y,x)) } ;
      |  w_leq-def(x,y)
      |
      +-I-node [1^3]
        | < subst-add
        |     [y <-- 0, x <-- s(x)] [x <-- 0] [y <-- s(y), x <-- s(x)] >
        |
        +-G-node [1^4]
        |    { def not(less(0,s(x))) } ;
        |    w_leq-def(s(x),0)
        |
        +-G-node [1^3:2]
        |    { def not(less(y,0)) } ;
        |    w_leq-def(0,y)
        |
        +-G-node [1^3:3]
          |  { def not(less(s(y),s(x))) } ;
          |  w_leq-def(s(x),s(y))
          |
          +-I-node [1^3:3:1]
            | < axiom-rewrite 1 [1^2] less-3 1  [x <-- y , y <-- x] >
            |
            +-G-node [1^3:3:1^2]
                { def not(less(y,x)) } ;
                w_leq-def(s(x),s(y))
```

as displayed by the command interpreter. Obviously, the goal at position 1^2 should be applied to the goal at position $1^3.3.1^2$ as induction hypothesis, but due to what was said about the inference rule `ind-subs` above, the prover does not allow this application of `ind-subs`. Now in executing the command

```
assign-name not-less-def [1:1]
```

QuodLibet turns the subtree of `leq-def` at position 1^2 into a new proof state tree named `not-less-def`, and replaces this subtree in the original proof state tree `leq-def` by a reference to the new proof state tree `not-less-def`. As a consequence, the user may invoke the desired inductive application of the goal

```
{ def not(less(y,x)) } ;
w_leq-def(x,y)
```

which labels the root of a proof state tree now, to the goal at position $1.3.1^2$ — in the *new* proof state tree, however — by entering the command

```
apply ind-subs not-less-def [ ] not-less-def [1:3:1:1]
```

Note that the last two parameters are necessary to indicate the exact position of the application of `ind-subs` in the current proof state graph (see below).

So far we have not dealt with the question of how the user can *navigate* through the current proof state graph when working with the command interpreter of QuodLibet. Navigation may be required e.g. for *focussing* on certain proof state trees or even goal nodes, possibly because the proof problems they represent are thought to be crucial to the success of the overall proof construction. Stated more precisely, we have to explain how the command interpreter determines the proof state tree and/or goal node which a command such as e.g. `apply` or `assign-name` is to be applied to.

Each command whose execution involves an existing proof state tree T has an *optional* formal parameter for T, which is denoted by *PS-Tree-Ident* in the syntax definition of the command language in Appendix B. Moreover, if the command also concerns a goal node ν in T then there is a second optional formal paramater for ν in the command, namely *GNode-Position*. For instance, in the above `apply` command the actual parameters `not-less-def` (for the proof state tree) and `[1:3:1:1]` (for the position of the goal node) occur, which correspond to these optional formal parameters. Now given that the user does not supply an actual parameter for *PS-Tree-Ident* in a command, which is usually the case for proof constructions carried out with the command interpreter, the command refers to the *current proof state tree*, i.e. the value of the system variable `current PS-tree` (see above); and when no actual parameter for *GNode-Position* is given, the value of the system variable `current G-node` for the current goal node of the concerned proof state tree is used by the command interpreter. In order to allow the user to change the values of these system variables, the command language provides the `set current PS-tree` command and the `set current G-node` command.

The last command sketched in this subsection is the `set weight` command. Recall that whenever a proof state tree T is created for a conjecture Γ, e.g. with a `prove` or an `assume` command (see above), a *weight variable* $\mathtt{w\_}T(x_1, \ldots, x_n)$ is generated for the weights of the goals labeling the goal nodes of T where $x_1, \ldots, x_n$ are the constructor variables occurring in Γ. Now assuming that T is applied as an induction hypothesis, the order subgoal resulting from this application of Inductive Subsumption or Inductive Rewriting (see Section 5.3.2) cannot be verified until all the occurrences of the weight variable $\mathtt{w\_}T$ in the current proof state graph have been replaced *consistently* with concrete weights (see Definition 4.2.1). The user can achieve this by entering a `set weight` command for the proof state tree T. A typical example illustrating how this command may — and in what phase of a proof construction it should — be used is given in Section 7.3.4. Note that the command interpreter cannot execute a `set weight` command for a proof state tree that is the subtree of another proof state tree, because a proof state tree created with an `assign-name` command (see above) inherits the weight variable of the original proof state tree. Moreover, no inference rule can be applied to a goal $\langle \Gamma\, ;\, w \rangle$ if a weight variable occurs in Γ, which may be the case if Γ is an order subgoal.

7.3.3 Miscellaneous Commands

This subsection presents a brief summary of the remaining commands of the command language with the exception of those necessary for working with the proof control unit[16] (see Chapter 8). Essentially, these commands constitute user amenities; they are not really required for the formalization of data types nor for the construction of proof state trees, but they significantly add to the user-friendliness of the command interpreter of QuodLibet.

The `initialize` command allows the user to remove all objects from the data units of the inference machine (see Figure 7.1), thereby restoring the *initial* state of the inference machine. Furthermore, the user can save the complete *current* state of the inference machine to a file using the `save IM-state` command, whereas a previously saved state can be restored from a file with the `restore IM-state` command.

Like other inductive theorem provers, QuodLibet also provides a "replay" facility, which is based on the command interpreter and can be invoked with the `execute` command. This command expects the name of a text file comprising QuodLibet commands. The commands are read in and executed by the command interpreter sequentially. Unless the keyword `echo` occurs as the last word in the command, the usual output of the command interpreter is suppressed (see Section 7.3.4). Note that the user can specify the directory which the system searches for the indicated command file in a `set script-directory` command.

Various kinds of textually presented information on the state of the inference machine can be accessed by the user through the `display` command. We refer to Appendix B for a complete list of the keywords that describe the information the user may request

[16] i.e. `compile`, `load`, `unload` and `call`

from the command interpreter. Observe that the command for displaying the proof state tree `leq-def` (see above) was

```
display PS-tree leq-def I-nodes
```

where the keyword `I-nodes` indicates that all of the information associated with the inference nodes of the proof state tree is requested, i.e. for each inference node the applied inference rule together with the actual parameters.

Mainly because of our experience that (initial) specifications or conjectures are often erroneous, we decided to add the `delete` command to the command language, even though the use of this command usually results in a loss of inductive theorems. The `delete` command allows the user to remove (i) defining rules, (ii) whole proof state trees or (iii) just subtrees of proof state trees that are specified by inference nodes, from the current proof state graph. Besides, the command language also provides the `reset weight` command which can be used to reintroduce weight variables for proof state trees whose weight variables were (inadequately) set before. Of course, QuodLibet ensures that after the execution of such a command, the inference machine is in exactly the *consistent* state which would have been reached if the deleted object had never been generated before or if the reintroduced weight variable had never been assigned a concrete weight before.

7.3.4 An Example of an Interactive Proof Construction

We conclude the section on the command language of QuodLibet by presenting a small example of an interactive proof construction. In the following record of a sample session with the command interpreter, the construction of a proof state tree for the "domain lemma" for the division operation $\text{def}(\mathsf{div}(x,y)) \lor y = \mathtt{0}$ (see Example 4.3.1) is shown. Note that a command entered by the user is always preceded by the system prompt QL[n].

```
            QuodLibet Command Interpreter
               Version 1.1  (June 1999)


QL[1] execute "div-spec"

Executing command file ~/div-spec.ql ...

Finished executing command file ~/div-spec.ql
```

In order to keep the example session to a reasonable length, we begin by executing the command file `div-spec.ql`. This file comprises (i) commands for defining the required sorts `Bool` and `Nat` and operators `less`, `minus` and `div`[17] as well as (ii) commands for introducing and proving the two "domain lemmas" `less-def` and `minus-def`:

```
QL[2] display conjectures

less-def (proved):
{ def less(x,y) }

minus-def (proved):
{ def minus(x,y),
  less(x,y) = true }


QL[3] prove { def div(x,y), y = 0 } div-def

Creating the PS-tree div-def

The current PS-tree is div-def

The current G-node in div-def is

G-node root  "div-def"
 { def div(x,y),
   y = 0 } ;
 w_div-def(x,y)
** current **

Done
```

Due to the pattern of the condition literals in the defining rules for div, we prove the conjecture `div-def` with a case analysis that is caused by the literal $\mathsf{less}(x, y) = \mathsf{true}$. The two subgoals resulting from the following application of Literal Addition represent the claims to be shown in the base case and in the induction step, respectively.

```
QL[4] apply lit-add less(x,y) = true.

Applying lit-add to root in div-def
results in two new subgoals

G-node [1^2]
 { less(x,y) =/= true,
   def div(x,y),
   y = 0 } ;
 w_div-def(x,y)
** current **
```

[17]as presented in Section 7.3.1

```
G-node [1:2]
 { less(x,y) = true,
   def div(x,y),
   y = 0 } ;
 w_div-def(x,y)

The current G-node in div-def is [1^2]

Done
```

The base case can be proved straightforwardly as follows:

```
QL[5] apply axiom-rewrite 2 [1] div-1 1 [ ]

Applying axiom-rewrite to [1^2] in div-def
results in one new subgoal

G-node [1^4]
 { less(x,y) =/= true,
   def 0,
   y = 0 } ;
 w_div-def(x,y)
** current **

The current G-node in div-def is [1^4]

Done


QL[6] apply def-decomp 2

Applying def-decomp to [1^4] in div-def
results in no further subgoals

The current G-node in div-def is [1:2]

Done
```

The subtree corresponding to the base case is "solved" at this point, and the goal node at position 1.2 becomes the current goal node. In the subsequent induction step, we first rewrite the clause of the current goal node (i.e. the induction conclusion) with the "recursive" defining rule `div-2` for `div`:

```
QL[7] apply axiom-rewrite 2 [1] div-2 1 [ ]

Applying axiom-rewrite to [1:2] in div-def
results in two new subgoals
```

```
G-node [1:2:1^2]
 { def less(x,y),
   less(x,y) = true,
   def div(x,y),
   y = 0 } ;
 w_div-def(x,y)
** current **

G-node [1:2:1:2]
 { ~def less(x,y),
   less(x,y) = true,
   def s(div(minus(x,y),y)),
   y = 0 } ;
 w_div-def(x,y)

The current G-node in div-def is [1:2:1^2]

Done
```

Note that `~def less(x,y)` is the QuodLibet notation for the literal $\neg\mathrm{def}(\mathsf{less}(x, y))$. The "domain lemma" `less-def` (see above) subsumes the goal at position $1.2.1^2$:

```
QL[8] apply lemma-subs less-def [ ]

Applying lemma-subs to [1:2:1^2] in div-def
results in no further subgoals

The current G-node in div-def is [1:2:1:2]

Done


QL[9] apply def-decomp 3

Applying def-decomp to [1:2:1:2] in div-def
results in one new subgoal

G-node [1:2:1:2:1^2]
 { def div(minus(x,y),y),
   ~def less(x,y),
   less(x,y) = true,
   def s(div(minus(x,y),y)),
   y = 0 } ;
 w_div-def(x,y)
** current **

The current G-node in div-def is [1:2:1:2:1^2]
```

```
Done
```

Now we can make subsumptive use of the conjecture `div-def` as induction hypothesis:

```
QL[10] apply ind-subs div-def [ x <-- minus(x,y) ]

Applying ind-subs to [1:2:1:2:1^2] in div-def
results in two new subgoals

G-node [1:2:1:2:1^4]
 { def minus(x,y),
   def div(minus(x,y),y),
   ~def less(x,y),
   less(x,y) = true,
   def s(div(minus(x,y),y)),
   y = 0 } ;
 w_div-def(x,y)
** current **

G-node [1:2:1:2:1^3:2]
 { w_div-def(minus(x,y),y) < w_div-def(x,y),
   ~def minus(x,y),
   def div(minus(x,y),y),
   ~def less(x,y),
   less(x,y) = true,
   def s(div(minus(x,y),y)),
   y = 0 } ;
 w_div-def(x,y)

The current G-node in div-def is [1:2:1:2:1^4]

Done
```

The goal at position $1.2.1.2.1^4$ is generated because the substitution $\{x \leftarrow \mathsf{minus}(x,y)\}$ gives rise to a definedness condition (see Definition 5.3.1 and Figure 5.14). This goal is subsumed by the "domain lemma" `minus-def` (see above):

```
QL[11] apply lemma-subs minus-def [ ]

Applying lemma-subs to [1:2:1:2:1^4] in div-def
results in no further subgoals

The current G-node in div-def is [1:2:1:2:1^3:2]

Done
```

The only *open* goal node at this point is the one at position $1.2.1.2.1^3.2$. It is labeled with the *order subgoal* arising from the application of `div-def` as induction hypothesis.

However, the order subgoal still contains occurrences of the weight variable `w_div-def`. Before we can prove it, we first have to assign a weight to `w_div-def`. Since the first argument of div decreases in every recursive call, the variable x is a suitable weight for the conjecture $\mathrm{def}(\mathsf{div}(x, y)) \vee y = \mathsf{0}$.

```
QL[12] set weight x

The current G-Node in div-def is

G-node [1:2:1:2:1^3:2]
 { minus(x,y) < x,
   ~def minus(x,y),
   def div(minus(x,y),y),
   ~def less(x,y),
   less(x,y) = true,
   def s(div(minus(x,y),y)),
   y = 0 } ;
 x
** current **

Done
```

Obviously, this order subgoal is subsumed by the *induction lemma* for the destructor minus (see Section 4.3), which we introduce through the following `assume` command:

```
QL[13] assume { minus(x,y) < x, less(x,y) = true, y = 0 } minus-ind-lma

Creating the PS-tree minus-ind-lma

The current G-node in minus-ind-lma is

G-node root  "minus-ind-lma"
 { minus(x,y) < x,
   less(x,y) = true,
   y = 0 } ;
 w_minus-ind-lma(x,y)
** current **

Done
```

Since the `assume` command does not affect the system variable `current PS-tree` for the current proof state tree (as opposed to the `prove` command), we can complete the construction of the proof state tree `dev-div` without using the optional parameter for this proof state tree (see Section 7.3.2) in the following `apply` command:

```
QL[14] apply lemma-subs minus-ind-lma [ ]
```

```
Applying lemma-subs to [1:2:1:2:1^3:2] in div-def
results in no further subgoals

The current G-node in div-def is root

Done
```

At this point the current proof state graph contains a proof attempt P for the root node of the proof state tree `div-def` which does not have any *open* goal nodes, i.e. $\mathrm{OGNd}(P) = \emptyset$.[18] However, P is not a *proof graph* for the root node of `div-def`, because P contains a *lemma node* for the conjecture `minus-ind-lma` which has not been proved yet (see Definition 6.1.8).

Hence for a complete proof of the conjecture `div-def` we still have to prove the assumed induction lemma for `minus`. In order to avoid explicit references to the proof state tree `minus-ind-lma` in the subsequent `apply` commands, we change the value of the system variable `current PS-tree` as follows:

```
QL[15] set current PS-tree minus-ind-lma

The current PS-tree is minus-ind-lma

The current G-node in minus-ind-lma is root
```

We omit the ensuing construction of the proof state tree `minus-ind-lma` in our presentation of the sample session except for the first apply command ("`QL[16]`") and the last one ("`QL[29]`"). Basically, we prove this conjecture using a straightforward structural induction with either x or y as induction variable.

```
QL[16] apply subst-add [x <-- 0, y <-- s(y)]
                       [y <-- 0]
                       [x <-- s(x), y <-- s(y)].

Applying subst-add to root in minus-ind-lma
results in three new subgoals

G-node [1^2]
 { minus(0,s(y)) < 0,
   less(0,s(y)) = true,
   s(y) = 0 } ;
 w_minus-ind-lma(0,s(y))
** current **
```

[18] This proof attempt P consists of (i) the entire proof state tree `div-def`, (ii) the root nodes of the proof state trees for the applied lemmas `less-def`, `minus-def` and `minus-ind-lma`, (iii) three $\mathcal{L}$-labeled arcs for the application of the lemmas, and (iv) one $\mathcal{I}$-labeled arc for the application of `div-def` as induction hypothesis.

```
G-node [1:2]
 { minus(x,0) < x,
   less(x,0) = true,
   0 = 0 } ;
 w_minus-ind-lma(x,0)

G-node [1:3]
 { minus(s(x),s(y)) < s(x),
   less(s(x),s(y)) = true,
   s(y) = 0 } ;
 w_minus-ind-lma(s(x),s(y))

The current G-node in minus-ind-lma is [1^2]

Done
```

⋮

The following application of the inference rule <-Decomposition completes the proof of the induction lemma for minus and thus the proof of the original conjecture `div-def`:

```
QL[29] apply <-decomp 1

Applying <-decomp to [1:3:1^7:2] in minus-ind-lma
results in no further subgoals

Proved inductive validity of minus-ind-lma
{ minus(x,y) < x,
  less(x,y) = true,
  y = 0 }

Proved inductive validity of div-def
{ def div(x,y),
  y = 0 }

The current G-node in minus-ind-lma is root

Done
```

At this point the current proof state graph contains proof graphs for each of the conjectures `less-def`, `minus-def`, `minus-ind-lma` and `div-def`. Consequently, due to Theorem 6.2.4 (2), *all* the clauses occurring in the four proof state trees are inductively valid w.r.t. current specification.

To summarize the proof construction carried out in this sample session we present the complete proof state tree `div-def` in the form used by the command interpreter for the `display` command:

```
QL[30] display PS-tree div-def I-nodes

+-G-node root  "div-def"
  |  { def div(x,y),
  |     y = 0 } ;
  |  x
  | ++ proved ++
  |
  +-I-node [1]
    | < lit-add less(x,y) = true >
    |
    +-G-node [1^2]
    | |  { less(x,y) =/= true,
    | |     def div(x,y),
    | |     y = 0 } ;
    | |  x
    | |
    | +-I-node [1^3]
    |   | < axiom-rewrite 2 [1] div-1 1 [ ] >
    |   |
    |   +-G-node [1^4]
    |     |  { less(x,y) =/= true,
    |     |     def 0,
    |     |     y = 0 } ;
    |     |  x
    |     |
    |     +-I-node [1^5]
    |        < def-decomp 2 >
    |
    +-G-node [1:2]
      |  { less(x,y) = true,
      |     def div(x,y),
      |     y = 0 } ;
      |  x
      |
      +-I-node [1:2:1]
        | < axiom-rewrite 2 [1] div-2 1 [ ] >
        |
        +-G-node [1:2:1^2]
        | |  { def less(x,y),
        | |     less(x,y) = true,
        | |     def div(x,y),
        | |     y = 0 } ;
        | |  x
        | |
        | +-I-node [1:2:1^3]
        |    < lemma-subs less-def [ ] >
        |
```

```
+-G-node [1:2:1:2]
  |  { ~def less(x,y),
  |    less(x,y) = true,
  |    def s(div(minus(x,y),y)),
  |    y = 0 } ;
  |  x
  |
  +-I-node [1:2:1:2:1]
    | < def-decomp 3 >
    |
    +-G-node [1:2:1:2:1^2]
      |  { def div(minus(x,y),y),
      |    ~def less(x,y),
      |    less(x,y) = true,
      |    def s(div(minus(x,y),y)),
      |    y = 0 } ;
      |  x
      |
      +-I-node [1:2:1:2:1^3]
        | < ind-subs div-def [x <-- minus(x,y)] >
        |
        +-G-node [1:2:1:2:1^4]
        | |  { def minus(x,y),
        | |    def div(minus(x,y),y),
        | |    ~def less(x,y),
        | |    less(x,y) = true,
        | |    def s(div(minus(x,y),y)),
        | |    y = 0 } ;
        | |  x
        | |
        | +-I-node [1:2:1:2:1^5]
        |    < lemma-subs minus-def [ ] >
        |
        +-G-node [1:2:1:2:1^3:2]
          |  { minus(x,y) < x,
          |    ~def minus(x,y),
          |    def div(minus(x,y),y),
          |    ~def less(x,y),
          |    less(x,y) = true,
          |    def s(div(minus(x,y),y)),
          |    y = 0 } ;
          |  x
          |
          +-I-node [1:2:1:2:1^3:2:1]
             < lemma-subs minus-ind-lma [ ] >

Done
```

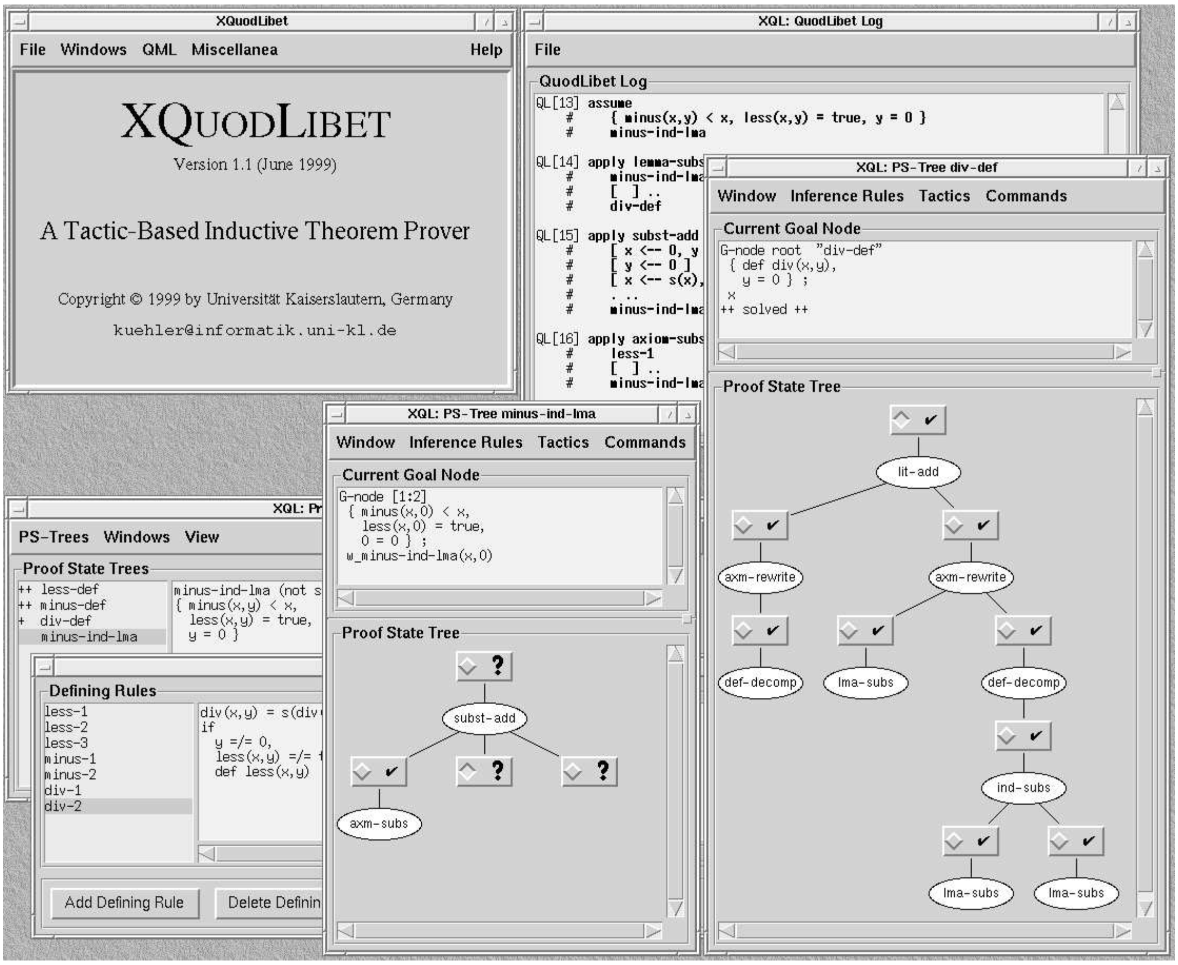


Figure 7.2: A session with XQuodLibet

7.4 The Graphical User Interface of XQuodLibet

Although it is possible to carry out (interactive) proof constructions solely by working with the command interpreter of QuodLibet, users may typically prefer the more convenient access to the functionality of the system through the windows, menus and other devices of XQuodLibet's graphical user interface (GUI). In this section we give a rough overview of the GUI of our inductive theorem prover by concentrating on its essential properties. From the point of view of the operating system, XQuodLibet consists of a LISP process representing the inference machine, the proof control unit and the command interpreter (see Figure 7.1) and of a Tcl/Tk process for the GUI. These two processes communicate with each other via their standard input/output streams. The concept for coupling LISP and Tcl/Tk processes which underlies XQuodLibet was developed and implemented by Jürgen Schumacher (see Schumacher & Kühler, 1997).

Figure 7.2 conveys a first impression of the various kinds of windows provided by the

GUI that are typically involved in proof constructions with XQuodLibet. Obviously, the sample session whose current state is presented in Figure 7.2 corresponds to the session discussed in Section 7.3.4. Note that two opened *proof state tree windows* belong to the depicted state: (i) the one in the lower right hand corner for the completed construction of the proof state tree `div-def` and (ii) the smaller one in the middle for the conjecture `minus-ind-lma`, where the first out of two base cases has just been proved.

The top-level window of XQuodLibet's GUI, which appears after starting up the system, is shown in the upper left-hand corner in Figure 7.2. The (pull-down) menus of this window[19] are compiled in the following figure.

Figure 7.3: Menus of the top-level XQuodLibet window

Except for the Extract Commmand File command (see below), all of the commands provided by the menus File, QML and Miscellanea evidently have direct counterparts in the command language, which were dealt with in Section 7.3 or will be introduced in Section 8.2.2. Moreover, the Windows menu allows the user to open the major windows that are needed for axiomatizing data types and constructing inductive proofs either by using the GUI or the command interpreter, which is integrated into the GUI of the system. We briefly explain the functions of these five windows in the following.

The windows QuodLibet Workspace and QuodLibet Log serve as the input and output windows of the command interpreter, respectively. Their use is illustrated in Figure 7.4, which shows a part of the sample session presented in Section 7.3.4. Within the so-called *workspace* of the command interpreter, the user can enter and conveniently edit commands of the QuodLibet command language (see Figure 7.4). Text contained in the workspace can be sent to the command interpreter by selecting it with the mouse and pressing *Ctrl-Return*. If no text is selected, pressing *Ctrl-Return* causes the GUI to send the contents of the line containing the cursor to the command interpreter. In the QuodLibet Log window, the system repeats every command entered in the workspace and presents the output produced by the command interpreter while executing the command (see Figure 7.4). Observe that for every command invoked through the GUI, XQuodLibet generates the corresponding command of the command language and displays it in the log window, which can be seen in Figure 7.2. Note also that the system offers the Extract Commmand File command for saving the commands recorded in the log window into a command file.

[19] without the menu of the Help facility though

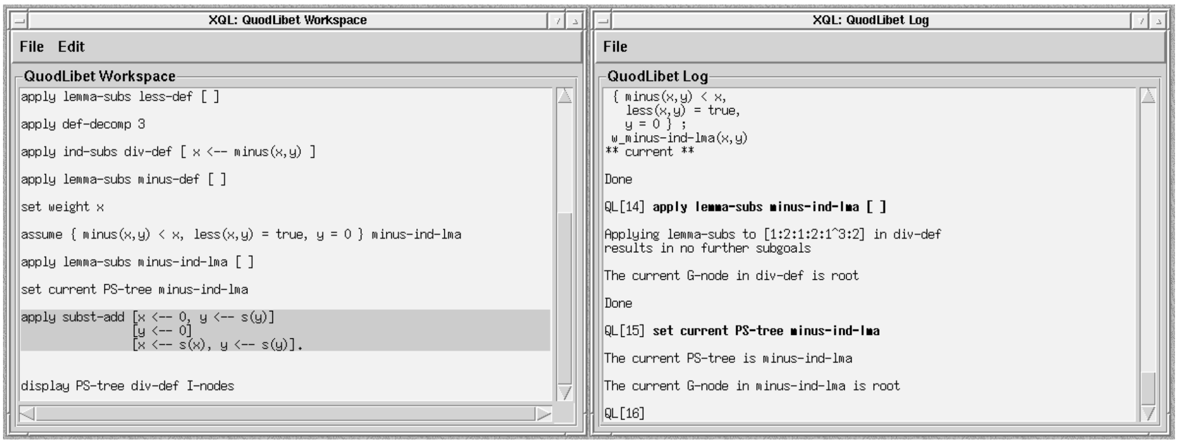


Figure 7.4: The windows for the command interpreter

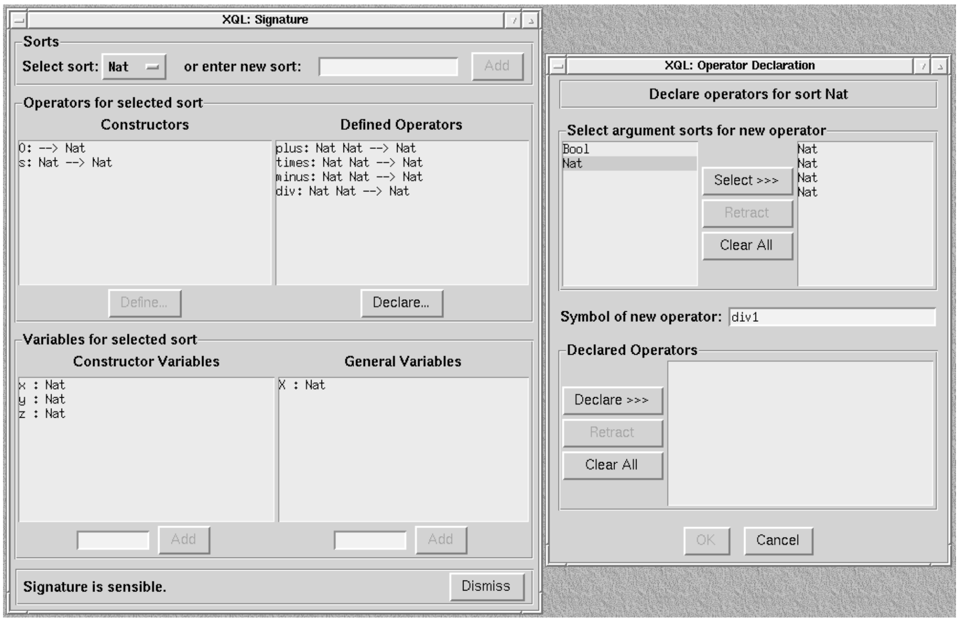


Figure 7.5: Declaring an operator with the GUI

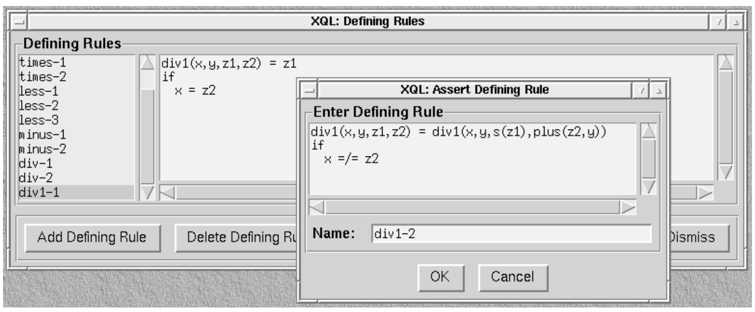


Figure 7.6: Introducing a defining rule with the GUI

On the one hand, the Signature window, i.e. the third window mentioned in the Windows menu (see Figure 7.3), is to inform the user about the signature that is currently stored in the inference machine. On the other hand, this window enables the user to extend the current signature by defining new sorts or declaring further (defined) operators and variables (refer to Section 7.3.1 for the corresponding commands of the command language). An example of the Signature window is depicted in Figure 7.5, together with a window in which the "non-terminating" operator div1 from Example 2.2.2 is being declared.

The Defining Rules window provides an overview of the set of defining rules that belong to the current state of the inference machine (see Figure 7.6). Moreover, this window can also be used (i) to add a single defining rule to the current set of defining rules and (ii) to remove a defining rule (i.e. the one selected) from the inference machine. The corresponding commands of the command language are the `assert` command (see Section 7.3.1) and the `delete` command (see Section 7.3.3). Note that in the smaller of the two windows in Figure 7.6, the user is just about to introduce the second defining rule for the operator div1, as it was given in Example 2.2.2.

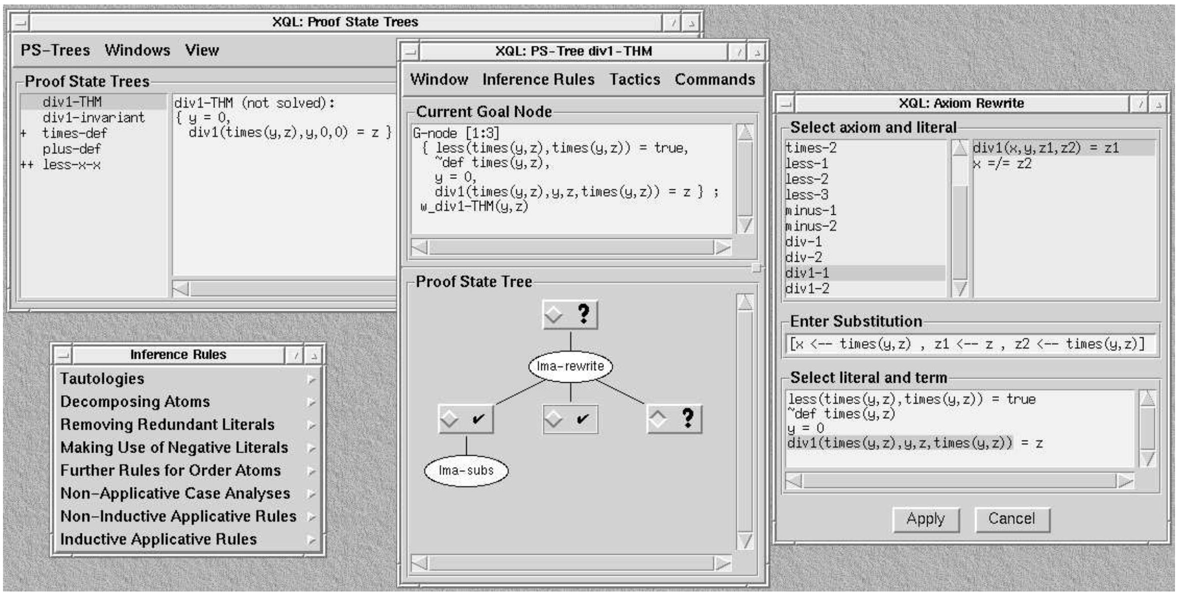


Figure 7.7: Applying an inference rule with the GUI

In the Proof State Trees window, the user can view the list of proof state trees contained in the current proof state graph. As the example Proof State Trees window in Figure 7.7 conveys, each name of a proof state tree in the left-hand subwindow is preceded by a so-called status indicator, which consists of a string of zero, one or two '+' characters. The string "++ T" means that the current proof state graph G contains a proof graph for the root node ν of the proof state tree T (i.e. the conjecture in the goal labeling ν is proved), while "+ T" indicates that there is no proof graph for ν in G yet, but a proof attempt P for ν in G exists which does not have any open goal nodes (i.e. a lemma node for an unproved conjecture must occur in P); and in case of an empty status indicator, each proof attempt for ν in G has some open goal node. Furthermore, the commands of the PS-Trees menu of the Proof State Trees window allow the user (i) to generate an initial proof state tree for a new conjecture, as can be done with the `prove` or `assume` command (see Section 7.3.2), and (ii) to delete an existing proof state tree (i.e. the one selected) in the current proof state graph.

Note that the proof state trees listed in the example Proof State Trees window in Figure 7.7 may arise in an attempt to prove the inductive validity of the clause (2.1) w.r.t. the specification $spec_{\mathsf{div1}}$ from Example 2.2.2. The clauses associated with these proof state trees are as follows:[20]

Proof state tree	Clause
`div1-THM`	$y = 0 \vee \mathsf{div1}(\mathsf{times}(y,z), y, 0, 0) = z$
`div1-invariant`	$y = 0 \vee \mathsf{less}(x, \mathsf{times}(y,z)) = \mathsf{true}$ $\vee\, \mathsf{div1}(x, y, 0, 0) = \mathsf{div1}(x, y, z, \mathsf{times}(y,z))$
`times-def`	$\mathrm{def}(\mathsf{times}(x,y))$
`plus-def`	$\mathrm{def}(\mathsf{plus}(x,y))$
`less-x-x`	$\mathsf{less}(x,x) = \mathsf{false}$

The window depicted in the middle of Figure 7.7 is a so-called *proof state tree window* for the proof state tree `div1-THM`. A proof state tree window, which can be opened for the selected proof state tree with the Windows menu of the Proof State Trees window, *visualizes* the (interactive) construction of a proof for the conjecture at the root of the proof state tree.

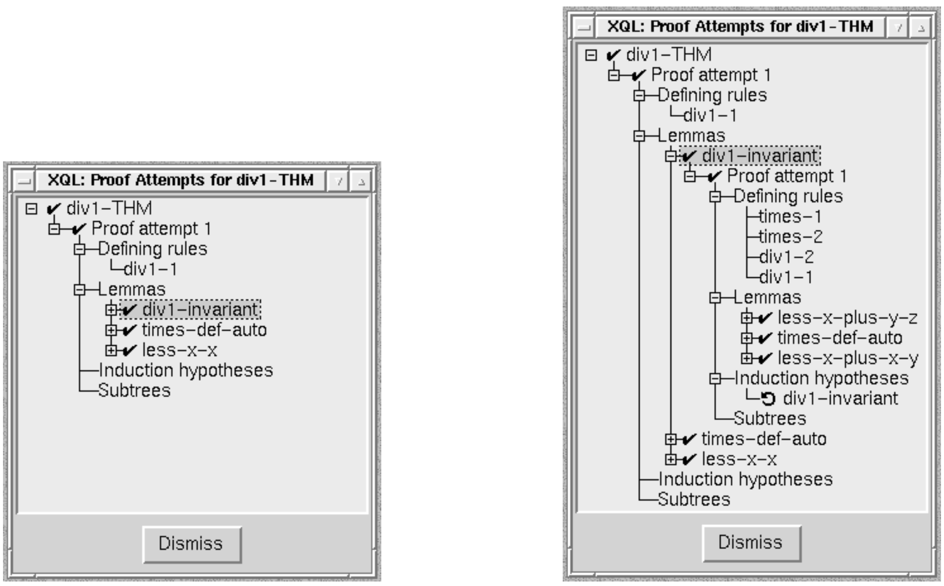
The lower of the two subwindows provides the graphical representation of the proof state tree. Obviously, nodes presented as rectangles are goal nodes, while nodes in elliptical shape are inference nodes. When the user clicks on a goal node ν with the left button of the mouse, the value of the system variable `current G-node` for the given proof state tree is changed to ν (see Section 7.3.2). Clicking on any node with the middle button of the mouse causes a temporary window to appear that contains the information associated with the node. Observe that the check ($\surd$) in a goal node ν indicates that ν is *solved* in that there are no more open goal nodes in the subtree below ν. If a goal node is not solved (yet), it is marked with a question mark.

In order to enable the user to cope with the enormous size of proof state trees in "real-life" applications to a certain degree at least, proof state tree windows facilitate various forms of *compressed* graphical representations of proof state trees. For instance, by clicking on a goal node ν with the right button of the mouse, the user can *fold* the subtree rooting in ν, which means that this subtree is presented by the single modified *subtree* goal node ν. The node at position 1.2 in the proof state tree `div1-THM` e.g. is a subtree goal node and stands for a whole subtree, which comprises four nodes and does not have any open goal nodes (see Figure 7.7). Of course, proof state tree windows also allow the user to *unfold* subtrees that are represented by subtree goal nodes.

[20]This example is also dealt with in Section 5.3.1.

The upper subwindow in a proof state tree window displays the position and the goal, i.e. the clause and the weight (variable), of the current goal node in the concerned proof state tree. For an expansion of the current goal node by applying one of the twenty-seven inference rules that QUODLIBET provides, the user can select the desired inference rule in the Inference Rules menu of the proof state tree window, which is depicted in the lower left-hand corner of Figure 7.7. Having selected an inference rule, the user is prompted to supply the actual parameters needed to specify the intended application of the inference rule in a further window (refer to Section 7.3.2 for the corresponding `apply` command of the command language and to Appendix C for a complete description of the QUODLIBET inference rules). For instance, the window on the right-hand side of Figure 7.7 depicts how *conveniently* the five parameters of the inference rule Non-Inductive Rewriting (i.e. `axiom rewrite`) can be provided by the user in order to perform a rewrite step to the fourth literal in the clause of the current goal node with the first defining rule `div1-1` for the operator div1. Note that the required matching substitution is determined by XQUODLIBET automatically, once the user has selected the four other actual parameters.

The Commands menu of the proof state tree window includes (among others) commands that correspond to the `assign-name` and `set weight` commands of the command language (see Section 7.3.2), whereas the tactics made available by the proof control unit may be invoked in the Tactics menu (see Section 8.2.2).



(a) (b)

Figure 7.8: Overview of the closed proof attempt for `div1-THM`

We conclude this section on the graphical user interface of XQUODLIBET by shortly dealing with one of its further features, for which no corresponding command exists in the command language of QUODLIBET (yet). The Proof Attempts entry in the View menu of the Proof State Trees window allows the user to obtain a graphical overview of the "closed" proof attempts contained in a proof state tree, where a proof attempt P

for the root of a proof state tree in the current proof state graph is said to be closed if there are no open goal nodes in P. As Figure 7.8 (a) illustrates, this graphical overview is presented in the form of a window that displays the applied defining rules, lemmas and induction hypotheses as well as the named subtrees[21] for each closed proof attempt in the proof state tree. In order to view the proof attempts of the applied lemmas or induction hypotheses, the user may simply click on their names with the left button of the mouse. For example, clicking on `div1-invariant` in Figure 7.8 (a) causes the contents of this window to change to that depicted in Figure 7.8 (b), where the names of the defining rules, lemmas and the induction hypothesis occurring in the proof graph for `div1-invariant` are also displayed now.

7.5 Concluding Remarks

The introduction to QuodLibet in this chapter has made it clear that our inductive theorem prover may be employed as a *full-scale* proof checker: Firstly, the functionality of the system as provided e.g. in the QuodLibet command language is such that it allows the user (i) to easily formalize numerous practically relevant data types and (ii) to construct virtually any inductive proof with the prover that is feasible within the formal framework of Part I. In particular, QuodLibet supports navigation through proof state trees, and goal nodes may be expanded more than once, thus giving rise to choice points in proof state graphs which represent multiple proof attempts for the same conjecture (see Section 6.1). Secondly, certain systematic experiments with our prover would confirm our claim that QuodLibet guarantees the soundness of every successfully executed command required in the axiomatization of a data type or in proof constructions. To be more precise, the inference machine always checks the given admissibility conditions of the specification language or the applicability conditions of the inference rules. Thirdly, the graphical user interface of XQuodLibet significantly adds to the user-friendliness of the prover by visualizing interactive proof constructions. In addition to the GUI, the system also provides a sometimes more efficiently usable command interpreter for experienced users.

With these remarks in mind it is easy to see that XQuodLibet essentially meets the first three of the four major requirements for an inductive theorem prover which we discussed in Section 7.1. Note, however, that our prover would hardly be accepted as a genuine proof *assistant* by potential users, if the forms of proof control it supports merely comprised — as presented so far — completely user-guided or interactive proof control. How "tiring" completely user-guided proof control may appear was illustrated in the sample session presented in Section 7.3.4. There, each step taken in the construction of the proof state tree `div-def` was determined by the user, regardless of how obvious some applications of inference rules actually were.

[21] generated with the `assign-name` command; see Section 7.3.2

Chapter 8

Tactic-Based Proof Control in QuodLibet

For QuodLibet to feature more "intelligent" forms of proof control than completely user-guided proof control, we suggest a novel approach to *automatically* controlling the inference machine of QuodLibet during the construction of inductive proofs. This approach enables QuodLibet to prove various simpler inductive theorems without user intervention, as will be shown in this chapter. In view of the fundamental problems posed by the automation of inductive theorem proving, which lead to our appreciation of the user's *active* role in successfully applying an inductive theorem prover to non-trivial conjectures (see Section 2.2.3), we were obviously *not* interested in "hard-wired" automated proof control as given in the form of a single proof procedure. Instead we have sought to come up with a new solution to the problem of proof control that is characterized by a great deal of *flexibility*. Roughly stated, we associate the following of QuodLibet's outstanding properties with the term 'flexibility' in the context of proof control:[1]

- Our prover provides a variety of what may be called *inference operators* which are applicable on different proof technical levels. The inference operators of the lowest such level correspond to the inference rules of QuodLibet, whereas inductive proof strategies capable of automatically proving simple inductive theorems are considered the inference operators of the highest level.
- QuodLibet allows the user to extend this collection of inference operators or to modify existing ones (except for the basic inference rules of course).

In this second chapter of Part II, we introduce QuodLibet's approach to proof control and attempt to give sufficient evidence for our conviction that flexibility with regard to proof control as characterized above can actually be achieved on the basis of our so-called (proof) *tactics*. Technically speaking, these tactics are expressions formulated in a new proof control language named QML, which can be thought of as a high level

[1] For an introductory discussion of proof control and our related requirements refer to the last part of Section 7.1.

programming language especially adapted to the programmed control of QUODLIBET's inference machine.

Note that there have been other (mostly higher-order) theorem proving environments with tactic-based approaches to (partially) automating proof constructions, such as e.g. LCF (Gordon, Milner, & Wadsworth, 1979), HOL (Gordon & Melham, 1993), NUPRL (Constable, 1986), OYSTER-CLAM (Bundy, van Harmelen, Horn, & Smaill, 1990) or TIGER (Gerberding & Noltemeier, 1997). Our concept of a tactic, however, essentially differs from each of the ones underlying the just mentioned systems, as should become apparent in the following (to those familiar with these systems).

This chapter is organized as follows. After giving a motivating overview of our approach to proof control in 8.1, we deal with the essentials of the proof control language QML in 8.2. Moreover, a description of the XQUODLIBET commands related to QML is given. Thereafter (in 8.3), we present the tactics — or rather (QML) proof control routines — which we provide as part of QUODLIBET and which realize the required (partial) automation of the proof control of our inductive theorem prover.

8.1 The Overall Idea

Before covering the main features of our approach to flexible forms of interactive and automated proof control in greater detail in the ensuing sections, let us first convey the overall idea of the approach. Recall that Section 7.1 contains a motivating discussion of our requirements with regard to proof control.

A first essential characteristic of our tactic-based approach to the problem of proof control is that it lays down fundamental properties of the *software architecture*[2] of our inductive theorem prover, and thereby affects the later (system) design phase as part of the process of developing QUODLIBET. To be more precise, the proposed approach to proof control determines (i) the top-level decomposition of the system into the following three main components or subsystems, namely

- a subsystem for the user interfaces
- a *proof control unit* as well as
- an *inference machine*

and (ii) the data and control flow between these subsystems as presented in Figure 7.1 (see Section 7.2). In addition to this, our approach makes some general requirements with regard to the inference machine:

- All the data objects representing the current specification and the current proof state graph are to be stored in (data units of) the inference machine.

[2]as defined in Section 7.2

- For the proof control unit (see below) and the user interfaces subsystem to effect any activities of the system which actually bring about changes of the current specification or of the current proof state graph, these subsystems are compelled to generate calls of suitable operations offered as resources at the interface of the inference machine.
- In particular, the choice of operations provided by the inference machine at its interface (refer to Section 7.2 for an overview of these so-called *inference machine operations*) has to ensure that *every* expansion of a goal node in the current proof state graph be the result of an application of one of the QuodLibet inference rules.

An important consequence of these architectural requirements is that our approach to proof control does not necessitate the "verification" of tactics. That is, every single change of the current proof state graph which is brought about by the proof control unit during the execution of any tactic is *legal*[3] in the sense that the inference machine is required to admit only steps during the construction of inductive proofs that comply with the definition of a proof state graph (see Definition 6.1.2). In other words, the inference machine can be considered the "secure" kernel of the prover.

A second major component of our tactic-based approach to proof control is the so-called *proof control unit* (PCU). This subsystem of XQuodLibet constitutes the setting for the realization of the flexible forms of interactive and automated proof control as required in Section 7.1. Against the background of its high level system structure (see the discussion above), it should be clear that in the case of QuodLibet, (automated) proof control virtually means (automated) control of the inference machine during the construction of inductive proofs. Therefore, the basic purpose of the PCU may be seen as controlling the inference machine over *several*, in some cases even over a large number of proof steps. As will become evident in the following, the control of the inference machine by the PCU is *programmed*. That is to say, when a (higher-level) inference operator provided by the PCU, such as e.g. an inductive proof strategy, is being applied to a proof problem, the PCU executes a corresponding *proof control routine* or tactic, thereby generating calls of inference machine operations as specified in the proof control routine. For reasons to be explained below, we have developed a special proof control language named QML for implementing such proof control routines.

In order to roughly characterize the user's view of the proof control unit, let us briefly deal with the system functions which the PCU itself[4] makes available to the user. Observe that the corresponding commands of the QuodLibet command language are described in Section 8.2.2.

- The PCU includes a compiler allowing the user to translate definitions of (new) QML routines, i.e. proof control routines written in QML, into code executable by the PCU.

[3]but not necessarily *reasonable* of course

[4]This does not include the proof control routines that are provided as part of QuodLibet (see below), although strictly speaking, they also add to the functionality of the PCU.

- Moreover, the PCU offers the user system functions for loading/removing the code of previously compiled QML routines into/from the data unit of this subsystem as shown in Figure 7.1.

- Last but not least, the PCU can also be employed to execute the code of QML routines, as was mentioned above.

By making use of this functionality, the user may, at any time, alter the set of QML routines that are accessible through the PCU, and thus we do not regard these proof control routines as part of the PCU. Hence, the PCU (along with QML) represents merely the *setting* for realizing the automation of the proof control of QUODLIBET.

A third main feature of our tactic-based approach to proof control is given by the proof control language *QML* (*Q*UODLIBET *M*eta*L*anguage). Experiences by other authors, such as e.g. Boyer and Moore (1979, 1988), establish that developing an inductive theorem prover with sufficiently efficient automated proof control amounts to a complex undertaking that includes a considerable *programming* effort. Because of this, QML essentially resembles a high level programming language which, however, is especially adapted to the needs of programming the control of QUODLIBET's inference machine during the construction of inductive proofs. One of the major requirements to guide the development of QML was to create a language that is particularly suited for the (comparatively) easy and convenient implementation of proof control routines by a *competent* user. The subsequent properties of QML contribute to our attempt at meeting this requirement:[5]

- There are predefined data types for objects of the *object language* of QUODLIBET, namely terms, weights, literals, clauses, goals, substitutions, proof state trees etc. Moreover, QML routines may call upon certain inference machine operations, e.g. upon those for applying inference rules or those for generating new proof state trees.

- QML enables the user to conveniently define and make use of new (recursive) data types.

- Besides *tactics*, i.e. QML routines intended to expand goal nodes by applications of inference rules, QML also admits “proof control” routines that never actually effect a change of the current proof state graph. An example of such a QML routine could be one for analyzing (recursive) axiomatizations of operators.

- QML features a series of control structures including a conditional construct and various iteration facilities. Furthermore, (recursive) calls of QML routines may occur in QML routines.

- QML provides a useful mechanism for handling so-called *exceptions*, which e.g. may be due to the non-applicability of an inference rule during the execution of a tactic.

[5]Section 8.2.1 deals with the essentials of QML in more detail.

- Last but not least, QML requires a modular organization of the proof control routines to be made available to the user through the PCU. The QML compiler expects the source code of exactly one QML module in every input file.

A crucial consequence of these properties is that by means of our proof control language, we achieve an easily comprehensible *abstraction* from the details of the design and implementation of (the inference machine of) QuodLibet. Thus, QML allows not only the system developers, but virtually *any* user who is also familiar with the QuodLibet command language, to participate in developing the proof control of QuodLibet by modifying available QML modules or adding new ones.

It has already been emphasized that the PCU (on the basis of QML) constitutes merely the setting for implementing the desired flexible forms of interactive and automated proof control. Consequently, our approach to proof control would not be comprehensive without the QML modules which we provide as part of QuodLibet. These modules make up the final major component of our approach and represent the "intelligence" of our inductive theorem prover in that they realize the required (partial) automation of the proof control of QuodLibet. In order to convey a rough impression of the additional functionality of the system which results from the provided QML modules, we now sketch the so-called *public* QML routines of these QML modules, i.e. those QML routines that the user may explicitly invoke through the PCU.[6]

- There are QML routines for initializing, extending and viewing the (proof control) *data base*. Entries into this data base can be brought about (i) by a QML routine that analyzes the defining rules for a given (recursive) operator of the current specification and tries to determine its domain, or (ii) by a QML routine that classifies a given proved conjecture with regard to possible uses as a lemma.
- One of our major (QML) tactics may be applied to perform a complete inductive case analysis for the clause of a given goal (node), whereas the call of a related tactic yields a case analysis based on the expansion of just one given operator in the goal. Both tactics are guided by the information on the definition schemes of the involved (recursive) operators as stored in the data base.
- We provide tactics for various simplification tasks, such as (i) recognizing tautologies or removing redundant literals, (ii) simplifying clauses with axioms and lemmas by rewriting and subsumption, or (iii) proving goals whose "leading" literals are certain definedness or order atoms.
- Fully automated proof control during the construction of inductive proofs for simpler conjectures is to be achieved with our strategy tactic (a restricted, usually terminating variant is also provided), which integrates the just mentioned tactics into a generally applicable inductive proof procedure.

Note that Appendix E contains a collection of example proofs that convey a first impression of how positively these (strategy) tactics affect the interactive construction of inductive proofs with QuodLibet.

[6]Refer to Section 8.3 for a more detailed discussion of these eighteen public QML routines, twelve of which are (QML) tactics.

8.2 The Proof Control Language QML

Having given a rather general overview of our approach to flexible forms of interactive and automated proof control in the foregoing section, we are ready to discuss, in greater detail now, the major features of this approach in the remainder of this chapter. This section mainly deals with the proof control language QML (8.2.1), but it also contains a summary of the XQUODLIBET commands needed to make use of the system functions provided by the proof control unit (8.2.2). Since we have to concentrate on the essentials of QML in this thesis, the reader is asked to refer to the comprehensive description of QML by Schmidt-Samoa (1997). For the most part, QML was developed by the author of this thesis and by Christof Sprenger (1996), with a few useful improvements being due to Tobias Schmidt-Samoa. The (detailed) design and the implementation of the proof control unit were carried out and extensively documented by Tobias Schmidt-Samoa (1997).

8.2.1 Essentials of QML

For QML to facilitate easy and convenient implementations of effective proof control routines by a greater number of users (as discussed in Section 8.1), we decided to use the widely known programming language Pascal (Wirth, 1971) as the starting point for the development of our proof control language. Therefore, QML resembles an *imperative* higher level programming language; it should be easily comprehensible for anyone familiar with other Pascal-like languages. As a proof control language, QML provides a variety of features that are designed to aid the user in writing proof control routines for the programmed control of QUODLIBET's inference machine during the construction of inductive proofs. Since QML is a language at least as extensive as Pascal, our description of QML can neither be complete nor systematic in this thesis. Instead we restrict ourselves to a selection of interesting QML concepts and constructs, which will roughly be characterized in the following. Note that Appendix D contains a complete definition of the syntax of QML in EBNF.

Modules

In the preceding section we pointed out that implementing the proof control routines which realize the (partial) automation of the proof control of QUODLIBET involves a considerable amount of programming. So in order to encourage users to decompose larger collections of QML routines into parts of "manageable" size and to make this modular structure more explicit, QML provides a concept of a *module*. Furthermore, with every (QML) module creating its own name space for resources such as types, variables and routines, the concept of a module has the advantage of preventing accidental name collisions which would typically result from the use of a single name space for all QML routines by different users. Modules communicate with each other explicitly via export and import interfaces. Every module is required to be contained in a separate input file (to the QML compiler).

```
MODULE Proof-Strategies;

EXPORT
  ROUTINE standard-strategy, restricted-strategy, set-weights;
  PUBLIC standard-strategy, restricted-strategy, set-weights;
END EXPORT;

IMPORT
  ROUTINE Inductive-Case-Analyses: ind-case-analysis-aux, ... ;
          Simplification: simplify-goal, apply-axiom, ... ;
          Database: termination-witnesses;
           ...
END IMPORT;
/*-------------------
   Exported Routines
  -------------------*/

TACTIC standard-strategy (goal: GNode);
  ...
END standard-strategy;

TACTIC restricted-strategy (goal: GNode);
  ...
END restricted-standard-strategy;

PROCEDURE set-weights (conj: DNode);
  ...
END set-weights;
/*--------------------
   Auxiliary Routines
  --------------------*/

TACTIC stand-strat-aux (goal: GNode);
  ...
END stand-strat-aux;

FUNCTION order-subgoal-p(goal: GNode): Boolean;
  ...
END order-subgoal-p;

END Proof-Strategies.
```

Figure 8.1: The QML module `Proof-Strategies` (abridged)

To give an example, Figure 8.1 shows, in a strongly abridged form, the QML module `Proof-Strategies`.[7] This module is provided as part of QUODLIBET. It contains the code for the two strategy tactics that the user may invoke through the PCU, namely `standard-strategy` and `restricted-strategy`. In addition to these tactics, the module `Proof-Strategies` exports the QML routine `set-weights` (a procedure[8]) that tries to replace each occurrence of the weight variables in the given proof state tree with a suitable weight. Observe that the module `Proof-Strategies` comprises two further auxiliary or *local* QML routines (a tactic and a function), which are not exported and thus invisible to all the other QML modules. The strings "`/*`" and "`*/`" delimit comments.

In general, every QML module consists of an export interface, an import interface and a series of arbitrarily many type definitions, variable declarations and routine definitions (see Appendix D). The export interface of a module M lists the types, variables and routines defined or declared in M that are to be accessed or *imported* by other modules. The `PUBLIC` clause of the export interface contains the names of those exported routines which QUODLIBET is to make available to the user through the PCU. We call a routine like this a *public* routine. The import interface of a module M specifies which of the types, variables and routines exported by the other modules may be used inside M. For each of these imported resources, the import interface must include the name of the exporting module.

Data Types

Due to its origin in Pascal, QML is a *typed* language with static types, i.e. every QML object has a fixed and unique type. Hence, type checking at compile-time is possible. There are certain predefined (data) types in QML, as well as means of defining new types by the user. Besides the standard data types `Integer`, `Real`, `String` and `Boolean` known from other programming languages, QML provides the following predefined data types for objects of the object language of QUODLIBET:

Data type	**Explanation**
`Sort`	sorts (of the signature)
`Operator`	function symbols (of the signature)
`Term`	terms
`Literal`	literals
`Formula`	formulas (i.e. clauses)
`Substitution`	substitutions
`Position`	positions (in tree-like structures)
`Weight`	weights
`DNode`	axiom nodes or root nodes of proof state trees
`GNode`	goal nodes
`INode`	inference nodes

[7]Each occurrence of three consecutive points (i.e. "...") in Figure 8.1 signifies an omission.

[8]A QML routine is either a tactic, a procedure or a function.

Note that a set of constants and operators belongs to each of these data types, some of which will be dealt with below in connection with QML expressions and the (library) routines provided by the PCU (for a complete description refer to Schmidt-Samoa, 1997).

QML allows the user to create new data types (on the basis of the predefined data types) by means of *type constructors* for list types, structured types and enumerated types. Except for list types, all these user-defined types must be named by declaring names for them through type definitions in QML modules.

Given that T is a named type or a list type itself, a *list type* for lists of elements of T is denoted by "[T]". For example, the type definitions

```
TYPE IntList   = [Integer];
     RealTable = [[Real]];
```

define named types for lists of integers and lists of lists of real numbers, respectively.

The elements of a *structured type* in QML are structures of a fixed number of different kinds. The composition of each kind of structure R belonging to a structured type is determined by a constructor function whose argument types define the types of the components of R. In other words, a structured type may be seen as a *collection* of the usual (non-variant) record types known from imperative programming languages (see Appendix D). As a simple example of a structured type, consider the following type definition of a recursive data type for binary trees whose nodes are labeled with integers:

```
TYPE BinTree = Empty () |
               Node (label: Integer;
                     left, right: BinTree);
```

In addition to the type `BinTree`, this type definition (implicitly) defines the constructor functions of `BinTree`, namely

```
FUNCTION Empty (): BinTree;

FUNCTION Node (label: Integer; left, right: BinTree): BinTree;
```

and the predicates (i.e. `Boolean`-valued functions) of `BinTree`, namely

```
FUNCTION isEmpty (bintree: BinTree): Boolean;

FUNCTION isNode (bintree: BinTree): Boolean;
```

An *enumerated type* introduces a set of constants, which are sorted in ascending order. For example, the type definition

```
TYPE EstimatedQuality = Low | Medium | High;
```

creates the constants `Low`, `Medium` and `High` of type `EstimatedQuality`. Moreover, the expressions `Low < Medium`, `Low < High` and `Medium < High` evaluate to the (QML) constant `TRUE`.

Observe that QML neither provides array types nor pointer types. Especially in the context of proof control, where "dynamic" and relatively complicated recursive data types are typical, we consider the proposed list types and structured types to be more convenient and useful.

Routines

A (QML) routine is either a tactic, a procedure or a function, and normally it is defined by the user in a routine definition. Every routine definition consists of, among other things, the name of the routine followed by a series of formal parameters and a sequence of statements (see Appendix D). When a routine is called upon by another routine or by the user (in the latter case it must be public), its formal parameters are replaced with the actual parameters given in the call, and its sequence of statements is executed. There are two kinds of formal parameters in QML, which are known from other imperative programming languages: value parameters and reference parameters.
Reference parameters are marked with the keyword `REF` in routine definitions.

A *tactic* is a QML routine intended to effect several, in some cases even a large number of inferences in a given proof state tree. More technically speaking, the first formal parameter of a tactic must be of type `GNode`. Furthermore, a call of a tactic is said to *fail*, thus causing an exception, if the current proof state graph is not extended by at least one successful application of a QUODLIBET inference rule during the execution of the tactic. QML provides a total of twenty-seven predefined *elementary* tactics, each of which represents one of the QUODLIBET inference rule. For example, the tactics

```
TACTIC =/=-taut (goal: GNode; lit-nb: Integer);

TACTIC subst-add (goal: GNode; subs: [Substitutions]);
```

correspond to the inference rules $\neq$-Tautology and Substitution Addition, respectively.[9] Note that tactics may be called upon in other tactics, but neither in procedures nor in functions.

There are no significant differences between QML *procedures* and *functions* on the one hand, and the procedures and functions used in imperative programming languages on the other hand. Thus, a few remarks should suffice here. In function definitions, QML expects `RETURN` statements indicating suitable return values. More importantly, the PCU provides a variety of so-called *library* routines (functions and procedures) as part of the predefined data types for objects of the object language of QUODLIBET. For instance, the library function

[9]Such *tactic heads* can be easily derived for the other predefined tactics from the information given in Appendix C.

```
FUNCTION term-unifier (term-1, term-2: Term): Substitution;
```

computes the most general unifier of the two argument terms if it exists; otherwise the call of the function fails giving rise to an exception. An example of a library procedure is given by

```
PROCEDURE prove (conj: Formula; name: String);
```

which has the same effect as the `prove` command of the QuodLibet command language (see Section 7.3.2). The reader is asked to refer to (Schmidt-Samoa, 1997) for more information on the QML library of the PCU.

Statements

QML statements describe the actions to be performed by QML routines. Like other imperative programming languages, QML provides (i) the usual assignment statement, (ii) a conditional statement, (iii) various repetitive statements, (iv) the usual call statement (for activating named tactics or procedures) and (v) a `RETURN` statement (for terminating the execution of a routine and indicating the return value of a function). In addition to these, QML features (vi) a `TRY` statement and (vii) a `FAIL` statement for handling and creating exceptions, respectively.

Let us make a few remarks before dealing with the `TRY` and the `FAIL` statement below. The conditional statement available in QML is an `IF` construct admitting statement patterns such as e.g.

`IF` ... `THEN` ... ;... `ELSIF` ... `THEN` ... ;... ;... ;... `ELSE` ... ;... ;... `END IF;`

Furthermore, besides the commonly known `FOR`, `WHILE` and `REPEAT` statements, QML provides a `FOREACH` statement for iterations over the elements of a list, as well as a `LOOP` construct which simply executes its body (a sequence of statements) repeatedly. Observe that each of the QML iteration facilities permits non-local exits in the form of the `EXIT` statement (see Appendix D). With regard to the call statement, we would like to emphasize that QML routines may be called *recursively*.

A crucial feature of QML, which considerably adds to the suitability of QML as a proof control language, and for which there do not exist counterparts in other imperative languages such as Pascal, is the `TRY` statement for handling exceptions. An *exception* arises when (1) a call of a tactic fails — this includes the case of a non-applicable inference rule that is represented by an elementary tactic —, when (2) a call of a library function such as `term-unifier` fails (see above), or when (3) the `FAIL` statement is executed. Consider the following schematic example of a `TRY` statement

... ; `TRY` S_1 `FAILURE` S_2 `SUCCESS` S_3 `END TRY;` ...

where S_1, S_2 and S_3 denote QML statements. Moreover, suppose this `TRY` statement is part of the routine definition of a QML routine T. Now given that an exception occurs while S_1 is being executed, the execution of T does *not* abort — i.e. the exception is *handled* — but continues with the statement S_2 in the `FAILURE` "branch" of the `TRY` statement; and in case that the execution of S_1 does not give rise to an exception, the statement S_3 in the `SUCCESS` "branch" is executed. Consequently, in order for a routine T to handle all the exceptions that could possibly result from a statement S in the definition of T, the user needs to employ a `TRY` statement to *protect* S by placing it after the `TRY` keyword. For if S is not protected like this and an exception due to S arises, the execution of T will abort and the call of T will fail. The exception is then propagated upwards in the hierarchy of routine calls — until the exception is handled in a higher routine by a `TRY` statement or the highest routine (i.e. the one invoked by the user) aborts.

Note that the `TRY` statement facilitates the implementation of inference processes to a large extent. One important reason for this experience is the fact that, by protecting *attempts* to apply a given inference rule in a `TRY` statement, the user does not need to write QML code for testing (all) the applicability conditions of the inference rule: if the attempted application fails, the resulting exception is handled. This code would be redundant anyhow, since the inference machine of QUODLIBET *always* has to establish that all the applicability conditions of an inference rule are fulfilled before the current proof state graph can actually be extended in a corresponding expansion step.

Expressions

Generally speaking, an expression can be seen as a rule for calculating values. Expressions consist of operators and operands, and they occur in various kinds of statements, for instance on the right-hand side of assignment statements or as actual parameters in routine calls (see Appendix D).

Every (QML) *operand* has a fixed and unique type and may be a variable, a constant, a function call or a so-called prover object, i.e. an object of the object language of QUODLIBET. Operands of the standard data types are formed as in other imperative programming languages. Because of the list types provided (see above), QML admits list operands such as e.g. (i) `["alpha" | l]`, where `l` is a variable of type `[String]`, or (ii) `[1, 2, 3 | [4, 5 | []]]`, which denotes the list containing the integers 1 through 5. The elements of the predefined data types `Term`, `Weight`, `Literal`, `Formula`, `Substitution` and `Position` are called *prover objects*. QML allows prover objects to be written in their QUODLIBET command language representation enclosed in quotes. Given that the current signature is the one used in the sample session of Section 7.3.4, the following examples are valid QML operands:

- `'0'`, `'div(minus(x, y), y)'`, `'true'` (`Term`)
- `'(minus(x, y))'`, `'(x, s(y))'` (`Weight`)
- `'~def less(x, y)'`, `'y = 0'`, `'minus(x, y) < x'` (`Literal`)
- `'{ s(x) =/= s(y), x = y }'`, `'{ def div(x, y), y = 0 }'` (`Formula`)

- `'[ ]'`, `'[ x <-- minus(x, y), y <-- s(0) ]'` (Substitution)
- `'[1:1:1:2:1]'`, `'[1^3:2:1]'` (Position)

In addition to the logical, arithmetic and relational *operators* usually available in other imperative programming languages, QML provides a few useful operators applicable to list operands and prover objects, namely `AT`, `IN`, `++` and `MATCH`.

The operator `AT` can be used to extract components from tree- or list-like objects. For instance, the expression `'~def less(x, s(y))' AT '[1:2]'` evaluates to `'s(y)'`, since $\neg\mathrm{def}(\mathsf{less}(x, \mathsf{s}(y)))\,/(1.2) = \mathsf{s}(y)$. Moreover, the expression `gamma AT i` returns (i) the i-th literal of the clause `gamma` if `gamma` is of type `Formula`, (ii) the i-th literal of the clause belonging to the goal node `gamma` if `gamma` is of type `GNode` or (iii) the i-th element of the list `gamma` if the type of `gamma` is a list type.

The operator `IN` tests for list inclusion. To be more precise, the expression x `IN` l is legal (and yields a `Boolean` result) given that (i) the type T of x is predefined and l is of type `[`T`]` or (ii) the type of x is `Literal` and the type of l is `Formula`.

The operator `++` concatenates (i) two lists of the same type or (ii) two positions.

A further amenity provided by QML is the `MATCH` operator. This operator can be used to test whether a term, a weight or a literal matches a (more general) *pattern* term, weight or literal, respectively. Furthermore, values are assigned to certain variables occurring in the pattern if the test succeeds. The `MATCH` operator is applicable to either two terms, two weights or two literals. The first argument represents the pattern and may contain so-called *pattern variables*, i.e. QML variables preceded by a '`%`'. The second argument is the prover object to match the pattern. If the matching process is successful then the `MATCH` operator returns `TRUE` and the pattern variables are assigned the values as required by the match. Otherwise, the expression evaluates to `FALSE` leaving the pattern variables undefined. Let us illustrate the `MATCH` operator using the following examples.[10] Assume that `x`, `term-1` and `term-2` are QML variables of type `Term`, whereas `lit` is a QML variable of type `Literal`.

- `'s(%x)' MATCH 's(s(x))'` returns `TRUE` with `'s(x)'` assigned to `x`. Note that in this expression, `x` denotes a QML variable as well as a variable of the object language of QuodLibet.
- `'s(%x) = %x' MATCH 's(s(x)) = s(y)'` evaluates to `FALSE`.
- `'s(%x) = %x' MATCH 's(s(x)) = s(x)'` returns `TRUE` with `'s(x)'` assigned to `x`, because $(\,\mathsf{s}(\mathsf{s}(x)) = \mathsf{s}(x)\,) =_{\mathrm{lit}} (\,\mathsf{s}(x) = \mathsf{s}(\mathsf{s}(x))\,)$.
- `'%term-1 =/= %term-2' MATCH lit` yields the result `TRUE` if the value of `lit` is a negative equational literal of the form $t_1 \neq t_2$. In this case, either t_1 and t_2 — or t_2 and t_1 — are assigned to `term-1` and `term-2`, respectively.

[10] For a more precise description of the `MATCH` operator see (Schmidt-Samoa, 1997).

```
TACTIC prove-tautology (goal: GNode);
VAR
  numb-lits, i: Integer;
BEGIN
  numb-lits := length(formula-literals(gnode-formula(goal)));
  FOR i := 1 TO numb-lits DO
    TRY
      prove-taut-lit(goal,i)
    SUCCESS
      EXIT
    END TRY
  END FOR
END prove-tautology;
```

Figure 8.2: The tactic `prove-tautology`

Having introduced the essentials of our proof control language QML at this point, we can now present an example involving two simple, but realistic tactic definitions in order to give the reader an impression of what QML code may typically look like.

Example 8.2.1 A rather small example of a user-defined tactic is given in Figure 8.2. The tactic `prove-tautology` is exported by the QML module `Simplification` (see Section 8.3) as a public routine. Thus, it can be invoked by the user. As the name suggests, `prove-tautology` is capable of proving (almost) every goal containing a "tautology", i.e. an obviously inductively valid clause.[11] In order to enable the reader to understand this really simple tactic, we provide the following information on the QML routines that are called in `prove-tautology`: The library function `gnode-formula` extracts the clause belonging to a goal node from the given goal node, while `formula-literals`, another library function, computes a list of the literals occurring in the argument clause. The library function `length` determines the length of a list. More importantly, the routine `prove-taut-lit` is an auxiliary user-defined tactic, whose definition is shown in Figure 8.3.

Roughly speaking, the tactic `prove-taut-lit` performs a case analysis based on the literal `lit-i` of the clause belonging to the goal node `goal`. Depending on the form of `lit-i`, the tactic attempts to apply the appropriate inference rule(s) — or rather, the corresponding elementary tactic(s) — for establishing simple tautologies or decomposing atoms (see Sections 5.2.1 and 5.2.2). The semantics of the library functions occurring in the tactic definition can be described as follows: (1) `ctr-term-p(term-1)` yields `TRUE` if `term-1` is a constructor term. (2) `order-atom-p(lit-i)` evaluates to `TRUE` if `lit-i` is an order atom. (3) `solved-p(goal)` returns `TRUE` if there are no more open goal nodes in the subtree below `goal`. (4) `negate-literal(lit-j)` computes the negation of the literal `lit-j`. Observe that the calls of the tactics for the inference rules =-Decomposition, def-Decomposition and Complementary Literals, i.e. `=-decomp`, `def-decomp` and `compl-lit`, respectively, need not be protected inside `TRY` statements (see above), since all of them cannot fail.

[11] An example of an application of `prove-tautology` is presented in Section 8.2.2.

```
TACTIC prove-taut-lit (goal: GNode; i: Integer);
VAR
  lit-i, lit-j: Literal;
  term-1, term-2: Term;
  j: Integer;
BEGIN
  lit-i := goal AT i;
  IF '%term-1 = %term-1' MATCH lit-i THEN
    =-decomp(goal, i)
  ELSIF '%term-1 =/= %term-2' MATCH lit-i THEN
    IF ctr-term-p(term-1) AND ctr-term-p(term-2) THEN
      TRY
        =/=-taut(goal, i)
      END TRY
    END IF
  ELSIF 'def %term-1' MATCH lit-i THEN
    IF ctr-term-p(term-1) THEN
      def-decomp(goal, i)
    END IF
  ELSIF order-atom-p(lit-i) THEN
    TRY
      <-decomp(goal, i)
    FAILURE
      TRY
        <-taut(goal, i)
      END TRY
    END TRY
  END IF;
  IF NOT solved-p(goal) THEN
    j := 1;
    FOREACH lit-j IN formula-literals(gnode-formula(goal)) DO
      IF lit-i = negate-literal(lit-j) THEN
        compl-lit(goal, i, j);
        RETURN
      END IF;
      j := j + 1
    END FOR
  END IF
END prove-taut-lit;
```

Figure 8.3: The auxiliary tactic `prove-taut-lit`

8.2.2 Commands Related to QML

Subsequent to the presentation of the essential features of QML, we now address the question as to how the proposed proof control language is integrated into QUODLIBET. To be more precise, we are going to introduce, in the remainder of this section, the part of the functionality of our inductive theorem prover which we left out in Sections 7.3 and 7.4. This part relates to QML and constitutes the functionality of the proof control unit (see Section 8.1); it consists of system functions for (i) compiling QML modules, (ii) loading and removing the code of compiled modules and (iii) invoking public QML routines (tactics and procedures) in the course of constructing inductive proofs. As in Chapter 7, we are going to describe these system functions in terms of the QUODLIBET command language, before briefly explaining how the user can invoke them through XQUODLIBET's graphical user interface.

The QML compiler of the PCU can be invoked by the user with the `compile` command of the QUODLIBET command language. As the only parameter of this command, the command interpreter expects the name of a (QML) source file which contains the type definitions, variable declarations and routine definitions of exactly one (QML) module. Prior to compiling a module M, all of the modules that export resources to be imported by M must have been compiled and loaded (see below). For instance, in order to compile the module `Proof-Strategies` shown in Figure 8.1, the user may activate the compiler by entering the command

```
compile "Proof-Strategies"
```

The compilation succeeds if the compiler detects neither syntactic nor semantic errors in this module and if the modules mentioned in the import interface, namely `Database`, `Simplification`, `Inductive-Case-Analyses` etc., were loaded before. In this case, the code (and further information) generated as the result of the compilation is loaded into the system automatically, and the command interpreter produces the output[12]

```
Compiling QML file ~/QML/Proof-Strategies.qml ...

Finished compiling QML file ~/QML/Proof-Strategies.qml

Loading QML module Proof-Strategies

Tactics available
  standard-strategy
  restricted-strategy
Procedures available
  set-weights

Done
```

[12]The "available" tactics and procedures listed in the output are the *public* routines of the module `Proof-Strategies` (see Figure 8.1).

In case of an error, the compilation aborts and the command interpreter displays an error message that usually indicates the kind of the error and its approximate location in the source file.

In addition to the `compile` command, the command language provides the `load` command for merely loading — instead of compiling — (the code of) a *previously* compiled module into the system. This command is useful whenever a module is needed whose source file has not been changed since its last compilation. Note that immediately after QUODLIBET is started, there are no QML modules in the system, i.e. every QML module must have been loaded (or compiled) prior to its first use.

The user can remove (the code of) a previously loaded QML module from the system with the `unload` command of the command language.

In loading a successfully compiled QML module, QUODLIBET makes the public routines of this module available to the user, which he can then invoke with the `call` command. Recall from Section 8.2.1 that a tactic or a procedure is *public* if its name occurs in both the `EXPORT` and `PUBLIC` clauses of the export interface of a module. Moreover, it is required that each of the formal parameters of a public routine is a value parameter of a type predefined in QML. Thus, we ensure that the user can always supply *constants* of the appropriate types as the actual parameters of a routine call in a `call` command. When invoking a tactic T, the user has to provide the desired values for the formal parameters of T in the `call` command, beginning with the *second* formal parameter of T. A value for the *first* formal parameter of T, which must be of type `GNode` and stands for the goal node the tactic T is to be applied to, can e.g. be

- given by the user in the form of the name of a proof state tree and the position of a goal node in that tree *after* the other parameters in the `call` command or
- determined by the command interpreter by using the system variables `current PS-tree` and `current G-node` of the concerned proof state tree

(refer to the end of Section 7.3.2 for the discussion on *navigation* in the current proof state graph). As a consequence, applications of tactics with `call` commands strongly resemble applications of inference rules with `apply` commands.

In order to illustrate the use of the `call` command and the execution of a tactic, we provide the following simple (and somewhat contrived) example. Suppose the current signature is again the one used in the sample session of Section 7.3.4. In executing the command

```
prove { div(x,y) = y, s(x) < s(y), 0 =/= s(y), ~def div(x,0) } ex-thm
```

QUODLIBET creates the proof state tree `ex-thm` for a conjecture which is obviously a tautology (due to the third literal). Assume that `ex-thm` is the current proof state tree and that `root` is the current goal node in `ex-thm`. Now given that the QML module `Simplification` was (successfully) compiled and loaded before, the command

```
call prove-tautology
```

allows the user to apply the public tactic `prove-tautology` to `root` in `ex-thm` in order to prove this tautology (see Figure 8.2). Provided that the system variable `display-mode` was set to `detailed` (refer to Appendix B for the `set display-mode` command), the output of the command interpreter while executing this tactic is

```
Calling prove-tautology for root in ex-thm ...

Calling prove-taut-lit for root in ex-thm ...

Call of prove-taut-lit fails

Calling prove-taut-lit for root in ex-thm ...

Application of <-decomp to root in ex-thm fails

Application of <-taut to root in ex-thm fails

Call of prove-taut-lit fails

Calling prove-taut-lit for root in ex-thm ...

Applying =/=-taut to root in ex-thm
results in no further subgoals

Proved inductive validity of ex-thm
{ div(x,y) = y,
  s(x) < s(y),
  0 =/= s(y),
  ~def div(x,0) }

Call of prove-taut-lit succeeds

Call of prove-tautology succeeds


The current PS-tree is ex-thm

The current G-node in ex-thm is root

Runtime: 0.01 sec.

3 attempted application(s) of inference rules
1 successful application(s) of inference rules

Done
```

Observe that this output illustrates fairly well how the tactics `prove-tautology` and its auxiliary tactic `prove-taut-lit` (see Figure 8.3) work in this particular case. The first call of `prove-taut-lit` in `prove-tautology` fails, because the first literal $\mathsf{div}(x,y) = y$ does not match the pattern `'%term-1 = %term-1'`. The second call of `prove-taut-lit` fails as well: With $\mathsf{s}(x) < \mathsf{s}(y)$ being an order atom, `prove-taut-lit` successively tries to apply the inference rules $<$-Decomposition and $<$-Tautology, in both cases without success. However, the third call of `prove-taut-lit` succeeds, as the application of $\neq$-Tautology for the literal $0 \neq \mathsf{s}(y)$ proves the conjecture. Thus, the `EXIT` statement in the success "branch" of the `TRY` statement in `prove-tautology` is executed, and so the call of `prove-tautology` terminates successfully as well.

After presenting the system functions related to QML in terms of the command language of QuodLibet, we are now going to briefly explain how the user can make use of these functions through the windows, menus and other devices of XQuodLibet's graphical user interface (GUI). Recall that a rough overview of the GUI of our inductive theorem prover was given in Section 7.4.

Figure 8.4: The menu QML

The menu QML, which is depicted in Figure 8.4 and belongs to the top-level window of the GUI (see Figure 7.2), evidently offers the user commands corresponding to the `compile`, `load` and `unload` commands of the command language. In addition, this menu contains an entry for each loaded QML module that provides public *procedures* (such as e.g. `set-weights` in `Proof-Strategies`), and allows the user to invoke these.

For a "graphical" application of a tactic to a goal node ν in a proof state tree T, a proof state tree window for T is needed, as is shown in Figure 8.5. Unless ν is the current goal node of T, the user must click on ν in the graphical representation of T with the left button of the mouse, thus making ν become the current goal node of T (see Section 7.4). The desired tactic can then be selected in the menu Tactics of the proof state tree window, which is depicted in the lower left-hand corner of Figure 8.5. This menu consists of entries for all the loaded QML modules that provide public *tactics*. Once a tactic has been selected, its parameter window appears, which (i) prompts the user to supply the actual parameters of the intended call of the tactic and (ii) includes a Call button for the actual invocation of the tactic. Since the tactic `prove-tautology` of the example application presented in Figure 8.5 does not have any further formal parameters (besides the required first one), its parameter window merely comprises a Call and a Cancel button.

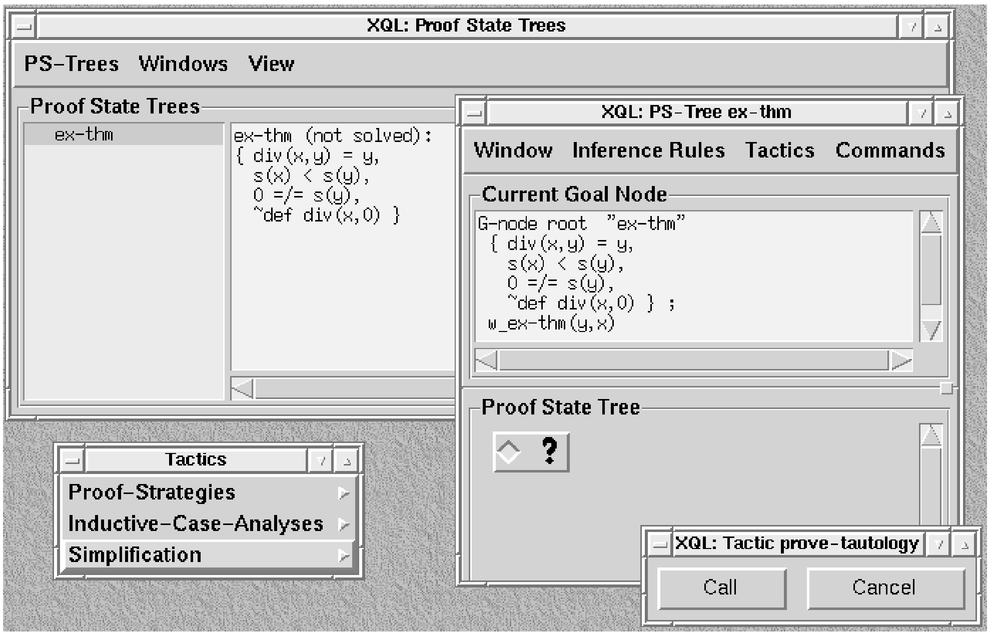


Figure 8.5: Applying a tactic with the GUI

8.3 QML Routines Provided by QUODLIBET

After giving a more thorough introduction to the proof control language QML in the preceding section, we are now going to deal with the final major feature of the proposed approach to proof control in greater detail, namely the QML modules provided along with our inductive theorem prover (see Section 8.1). In a first and "rapid" attempt to realize the required (partial) automation of the proof control of QUODLIBET, the author of this thesis has developed suitable concepts and proof control routines, mainly using ideas presented by Boyer and Moore (1979) and Walther (1994) as a guideline, and implemented these modules with the assistance of Christian Embacher. As the example proof constructions given in Appendix E suggest, the routines of these modules yield rather satisfactory experimental results in many cases. Nevertheless, we intend to re-address the problem of (semi-) automating proof constructions with QUODLIBET more *systematically* in the future when more time will be available. To this end, more of the relevant literature on the various (heuristic) approaches to the automation of inductive theorem proving will have to be taken into consideration, e.g. the ones reported by Bundy et al. (1993), Ireland and Bundy (1996), Hutter (1997), Kapur and Subramaniam (1996a, 1996b), or Bouhoula (1996).

On completion of this thesis, the following QML modules belong(ed) to QUODLIBET: (i) the three "auxiliary" modules `basics`, `substitutions` and `condition-trees`, that do not provide any public routines, and (ii) the four modules `Database`, `Simplification`, `Inductive-Case-Analyses` and `Proof-Strategies` that make a total of six public procedures and twelve public tactics available to the user. Because of the considerable size of these modules — all in all, they consist of more than 4,500 lines of

(QML) code and contain the definitions of approximately 130 routines besides various types —, we have to restrict the presentation of these modules to an explanation of the underlying general ideas and to a characterization of the additional functionality of the system which results from these eighteen public procedures and tactics. In this section, we focus on (i) the so-called (proof control) data base which stores and provides, among other kinds of information, the results obtained during the analysis of the (recursive) operators (8.3.1), and on (ii) the tactics we developed for generating inductive case analyses (8.3.3). Moreover, we sketch our tactics for simplifying clauses (8.3.2), and briefly describe our two strategy tactics (8.3.4).

8.3.1 The Proof Control Data Base

Undoubtedly, outstanding publications on inductive theorem proving such as the ones by Aubin (1976) or Boyer and Moore (1979), have made it clear that the success of any *heuristic* attempt to mechanize the construction of inductive proofs depends to a large extent on the availability of various kinds of information extracted from the underlying axiomatization and the current state of the proof construction. Hence, our approach to (partially) automating the proof control of QuodLibet employs a so-called *(proof control) data base* whose overall purpose is to manage relevant information to be used for guiding the construction of inductive proofs with the system. Roughly speaking, this proof control data base comprises (i) an operator table for the information obtained during the analyses of the defining rules for the defined (i.e. non-constructor) operators and (ii) a classified collection of the *activated* lemmas (see below). Note that the information stored in the data base is accessed whenever adequate inductive case analyses need to be found or clauses in goals have to be simplified, as will be explained in Sections 8.3.2 and 8.3.3. There are five public procedures related to the data base, each of which is made available to the user by the QML module `Database` and each of which will be briefly introduced in the following. In order to keep this section reasonably concise, we will not deal with the other (i.e. non-public) routines exported by this module, for instance those for retrieving information from the operator table or the collection of activated lemmas.

Initializing the Date Base

The sole purpose of the public procedure `initialize-database` provided by the QML module `Database` is to create the (proof control) data base in its initial state. Since the data base does not contain any data in its initial state, each invocation of this procedure[13] results in the loss of all the information currently stored in the data base. Observe that for a merely technical reason, a call of `initialize-database` must have been executed before any data can be added to the data base by the `analyze-operator` or `activate-lemma` procedures (see below).

[13]There is only one data base, and so `initialize-database` has no parameters.

Analyzing the Axiomatizations of Defined Operators

After introducing and axiomatizing a defined operator f (e.g. by using the command `define operator` described in Section 7.3.1), the user is expected to invoke the public procedure `analyze-operator` for f. We urge the user to actually do so because most of the other procedures and tactics yet to be presented in this section are based on the assumption that each defined operator f of the current specification has been analyzed prior to any proof construction involving f. During the analysis of the defining rules for f, `analyze-operator` tries to determine the so-called *definition scheme* of f and creates a corresponding entry into the operator table of the data base for f.

The data objects making up the definition scheme of f (see below) are to provide crucial information for the generation of *inductive case analyses* in proofs of conjectures about (the "behavior" of) f. Like the developers of other automated inductive theorem provers, we obtain inductive case analyses by using the well-known *induction heuristic.* The induction heuristic is based on the similarity of recursion and induction; it states that, roughly speaking, the case analysis used in the axiomatization of a terminating (and recursive) operator f often suggests a "promising" inductive case analysis for the proof of a conjecture Γ about f, i.e. a conjunction of formulas $\Gamma_1,\ldots,\Gamma_n$ representing the claims to be shown in the base cases and inductions steps for Γ, respectively. Instead of rephrasing explanations of the induction heuristic, which can be found e.g. in Chapter XIV of the book by Boyer and Moore (1979) or in Section 3.2 of the survey by Walther (1994), we give a simple example in order to convey an intuitive impression of the induction heuristic in the case of QUODLIBET.

Example 8.3.1 The following (recursive and conditional) definition axiomatizes an operator elem that tests for list inclusion.[14]

```
define operator  elem : Nat ListNat --> Bool
with defining rules
elem-1:
   elem(x,nil) = false
elem-2:
   elem(x,cons(y,l)) = true       if  x = y
elem-3:
   elem(x,cons(y,l)) = elem(x,l)  if  x =/= y.
```

In a way, the case analysis used in this definition may be represented by (i) the two constructor substitutions $\sigma_1 = \{l \leftarrow \mathsf{nil}\}$ and $\sigma_2 = \{l \leftarrow \mathsf{cons}(y,l)\}$ and (ii) the condition literal $x = y$, which gives rise to the two "σ_2-rules" `elem-2` and `elem-3`. Moreover, it is heuristically plausible to assume that elem is terminating (in an intuitive sense), which a simple comparison of the argument tuple (x,l) of the only recursive call of elem with the "greater" argument tuple $(x,\mathsf{cons}(y,l))$ of the corresponding call in the left-hand side of the defining rule `elem-3` shows.

Now suppose the conjecture about elem to be proved is the statement that the operator

[14] Refer to Section 7.3.1 for the underlying sort definitions and variable declarations.

elem is totally defined. A proof state tree for the proof of this “domain lemma” can be created with the command

```
prove { def elem(x,l) } elem-def
```

According to the induction heuristic (adapted to our formal framework for inductive theorem proving), the call of the recursive and terminating operator elem occurring in the conjecture may suggest an adequate inductive case analysis for a proof of `elem-def` that corresponds closely to the case analysis used in the definition of elem. To be more precise, we can construct this inductive case analysis for `elem-def` by employing the substitutions σ_1 and σ_2 on the one hand and the condition literal $x = y$ on the other hand as the parameters in two applications of the inference rules Substitution Addition[15] and Literal Addition. The inductive case analysis for `elem-def`, i.e. the proof state tree `elem-def` resulting from the two applications of these inference rules for non-applicative case analyses (see Section 5.2.6), looks as follows and is in fact well suited for a proof:

```
+-G-node root  "elem-def"
  |  { def elem(x,l) } ;
  |  w_elem-def(x,l)
  |
  +-I-node [1]
    | < subst-add
    |    [l <-- nil] [l <-- cons(y,l)] >
    |
    +-G-node [1^2]
    |    { def elem(x,nil) } ;
    |    w_elem_def(x,nil)
    |
    +-G-node [1:2]
      |  { def elem(x,cons(y,l)) } ;
      |  w_elem-def(x,cons(y,l))
      |
      +-I-node [1:2:1]
        | < lit-add
        |    x = y >
        |
        +-G-node [1:2:1^2]
        |    { x =/= y,
        |      def elem(x,cons(y,l)) } ;
        |    w_elem-def(x,cons(y,l))
        |
        +-G-node [1:2:1:2]
             { x = y,
               def elem(x,cons(y,l)) } ;
             w_elem-def(x,cons(y,l))
```

[15] $\{\sigma_1, \sigma_2\}$ is a cover set of substitutions for the goal of the root of `elem-def` (see Definition 5.2.14).

The three open goal nodes of `elem-def` (i.e. the leaves of the tree) represent the two base cases (the goals at positions 1^2 and $1.2.1^2$) and the induction step (the goal at position 1.2.1.2) in a proof of the conjecture. Note that the call of elem in each of these goals may now be *expanded*, i.e. rewritten with one of the defining rules for elem in an application of the inference rule Non-Inductive Rewriting. In particular, the goals at positions $1.2.1^2$ and 1.2.1.2 contain the (condition) literals needed for "safely" rewriting the calls of elem with the conditional defining rules `elem-2` and `elem-3`.[16]

As mentioned above, the essential information to guide the construction of an inductive case analysis for a conjecture about a defined operator f is to be provided by the so-called *definition scheme* of f. From the induction heuristic it follows that the definition scheme of f primarily has to serve the purpose of providing an adequate *representation of the case analysis used in the axiomatization of* f. A first idea of what (some of) the components of the definition scheme of elem might be was given in Example 8.3.1: the cover set of substitutions $\{\sigma_1, \sigma_2\}$ and the condition literal $x = y$ for the "σ_2-rules". We now explain in more detail what a definition scheme of f is composed of and how it is extracted from the defining rules for f by the procedure `analyze-operator`.

In what follows, we assume that (1) $sig = (S, F, \alpha)$ is the current signature, (2) $C \subseteq F$ is a set of constructors for sig and (3) $spec = (sig, C, R)$ is the current specification with constructors.

Besides the fundamental requirement that $spec$ be an admissible specification with free constructors, i.e. R must be a set of defining rules,[17] our analysis of the definition of a defined operator $f \in F \backslash C$ is based on the (rather natural) preconditions that the left-hand side of a defining rule for f must be a call of f whose arguments are *constructor* terms, and that "extra variables" in defining rules are not admitted.[18] Formally, the following properties are expected to be satisfied by each defining rule $l = r \leftarrow \Delta$ in R:

(a) There is a $f \in F \backslash C$ with $\alpha(f) = s_1 \ldots s_n s$ and $t_i \in \mathcal{T}(sig^C, V^C)_{s_i}$ for $i = 1, \ldots, n$ such that $l = f(t_1, \ldots, t_n)$.

(b) $\mathrm{Var}(l = r \leftarrow \Delta) \subseteq \mathrm{Var}(l)$

When invoked for a defined operator f, the first step to be taken by the procedure `analyze-operator` is to check whether the defining rules for f are *normalized* in the sense that, roughly speaking, as few variables as possible are employed consistently in the definition of f (see below). If this is not the case, variable renamings are used to compute normalized variants of some of the defining rules for f. Thereafter, `analyze-operator` partitions the normalized defining rules according to their left-hand sides. The result is an internal representation of the defining rules for f of the following form (assuming that $\alpha(f) = s_1 \ldots s_n s$)

[16] In this context, "safe" is to express that the application of (the `axiom-rewrite` variant of) Non-Inductive Rewriting leads to only one new subgoal (see Section 5.3.1 and Appendix C).

[17] see Definitions 3.2.1(b) and 3.2.4

[18] If any of these preconditions is not fulfilled by a defining rule for f, `analyze-operator` aborts the analysis of the definition of f.

$$\begin{array}{rcccc}
f(x_1\sigma_1,\ldots,x_n\sigma_1) & = & t_{1,1} & \leftarrow & \Delta_{1,1} \\
\vdots & & \vdots & & \vdots \\
f(x_1\sigma_1,\ldots,x_n\sigma_1) & = & t_{1,k(1)} & \leftarrow & \Delta_{1,k(1)} \\
 & & \vdots & & \\
f(x_1\sigma_m,\ldots,x_n\sigma_m) & = & t_{m,1} & \leftarrow & \Delta_{m,1} \\
\vdots & & \vdots & & \vdots \\
f(x_1\sigma_m,\ldots,x_n\sigma_m) & = & t_{m,k(m)} & \leftarrow & \Delta_{m,k(m)}
\end{array}$$

where (1) $x_i \in V^{\mathrm{C}}_{s_i}$ for $i = 1,\ldots,n$ (the "formal parameters" of f); (2) $\sigma_1,\ldots,\sigma_m$ are constructor substitutions such that $\sigma_i(x) = x$ if $x \notin \{x_1,\ldots,x_n\}$ for every $x \in V^{\mathrm{C}}$ and $i = 1,\ldots,m$; and (3) $t_{i,j} \in \mathcal{T}(sig,V)_s$ and $\Delta_{i,j}$ are clauses (the condition literals) for $i = 1,\ldots,m$ and $j = 1,\ldots,k(i)$. In the ensuing step, `analyze-operator` "completes" the set $\{\sigma_1,\ldots,\sigma_m\}$ in that it determines constructor substitutions $\sigma_{m+1},\ldots,\sigma_{m+p}$ such that $\{\sigma_1,\ldots,\sigma_m,\sigma_{m+1},\ldots,\sigma_{m+p}\}$ constitutes a cover set of substitutions for the goal $\langle\, \mathrm{def}(f(x_1,\ldots,x_n))\ ;\ ()\rangle$ (see Definition 5.2.14).[19] This is done on the basis of an algorithm for computing the (set theoretic) difference of sets of tuples of constructor terms.[20]

Let us illustrate the steps discussed so far with two examples.

Firstly, suppose the following definition is used to axiomatize the (partial) subtraction.

```
define operator  minus : Nat Nat --> Nat  with defining rules
minus-1:
  minus(x,0)        = x
minus-2:
  minus(s(y),s(z)) = minus(y,z).
```

Apparently, this definition is not normalized in the sense described above. Since the procedure chooses $x_1 = y$ and $x_2 = z$ as the formal parameters of minus, the first defining rule `minus-1` for minus is (internally) changed to $\mathsf{minus}(y,\mathsf{0}) = y$. Now the definition is normalized with $m = 2$, $\sigma_1 = \{z \leftarrow \mathsf{0}\}$ and $\sigma_2 = \{y \leftarrow \mathsf{s}(y), z \leftarrow \mathsf{s}(z)\}$.

[19] Of course, no substitutions are generated (i.e. $p = 0$) if the set $\{\sigma_1,\ldots,\sigma_m\}$ already has the desired cover set property.

[20] We would like to thank Wolfgang Lindner at this point. He made us aware of this algorithm, which he used in the implementation of Unicom (Gramlich & Lindner, 1991) for computing inductively complete positions, and pointed out to us that we could easily adapt it to the purpose of testing for completeness of cover sets of substitutions.

Then, `analyze-operator` checks whether $\{\sigma_1, \sigma_2\}$ is a cover set of substitutions for the goal $\langle\, \mathrm{def}(\mathsf{minus}(y, z))\ ;\ ()\rangle$, which yields the missing "case" $\sigma_3 = \{y \leftarrow \mathsf{0}, z \leftarrow \mathsf{s}(z)\}$ (i.e. $p = 1$).

Secondly, recall the destructor recursive definition of the operator div from Section 7.3.1. The defining rules for div are normalized so that x and y can serve as the formal parameters of div. Moreover, $m = 1$ and $\sigma_1 = \mathrm{id}$. Note that $\{\sigma_1\}$ is a *trivial* cover set of substitutions for the goal $\langle\, \mathrm{def}(\mathsf{div}(x, y))\ ;\ ()\rangle$ (i.e. $p = 0$).

Having determined the formal parameters and an associated cover set of substitutions for the defined operator f, the procedure `analyze-operator` applies a simple heuristic for *guessing* as to whether or not f is *terminating* (in an intuitive sense not to be defined here). Recall in this context that the soundness of our formal framework for inductive theorem proving does not presuppose any termination property of the defining rules in R (see Theorem 6.2.4, Definition 3.2.4 and Theorem 3.3.2). Instead, applications of induction hypotheses give rise to order subgoals which introduce the termination related proof obligations (see Sections 5.3.2 and 4.3). The essential reason for our interest in termination guesses is the termination requirement mentioned in the induction heuristic (see above). Consequently, if f is conjectured to be non-terminating, the computation of the definition scheme of f is aborted, as calls of f do not suggest inductive case analyses.

The termination heuristic used by the procedure `analyze-operator` is based on rather simple syntactic criteria and thus relatively crude. First of all, it is tested whether the defined operator f is *recursive*. We call f recursive if the function symbol f occurs in the right-hand side t or the condition literals Δ of a defining rule $f(t_1, \ldots, t_n) = t \leftarrow \Delta$ for f. In case that f is not recursive, f is conjectured to be terminating. Otherwise, `analyze-operator` tries to find so-called *termination witnesses* for f. To define the notion of a termination witness, we first have to introduce a (binary) relation $\leq_H$ on $\mathcal{T}(sig, V)$ for comparing arguments in function calls. We say that $t' \leq_H t$ if

(a) t' is a sub-term of t ; or

(b) there is a $d \in F \backslash C$ with $\alpha(d) = \bar{s}_1 \ldots \bar{s}_l \bar{s}$, $\bar{t}_i \in \mathcal{T}(sig, V)_{\bar{s}_i}$ for $i = 1, \ldots, l$ and a $j \in \{1, \ldots, l\}$ such that

  (i) $t' = d(\bar{t}_1, \ldots, \bar{t}_l)$; and

  (ii) $t = \bar{t}_j$; and

  (iii) the data base contains an (activated) induction lemma of the form

$$\langle\, d(z_1, \ldots, z_l) < z_j\, , \Sigma\, ;\, w\rangle$$

  where $z_i \in V^{\mathrm{C}}_{\bar{s}_i}$ for $i = 1, \ldots, l$ and Σ is a sequence of literals.[21]

We will explain below how an induction lemma for a destructor d (see also Section 4.3) can be *activated*, i.e. inserted into the data base with the procedure `activate-lemma`.

[21] i.e. d is a j-bounded algorithm in the terminology of Walther (1988)

By making use of $\leq_H$, we can now define termination witnesses. Let f be a recursive operator with $\alpha(f) = s_1 \dots s_n s$. A tuple of k distinct integers $i_1 \dots i_k \subseteq \{1, \dots, n\}^*$ is said to be a *termination witness* for f if for each defining rule $f(t_1, \dots, t_n) = t \leftarrow \Delta$ for f and for each (sub-) term of the form $f(t'_1, \dots, t'_n)$ occurring in t or Δ (i.e. a recursive call of f) the following condition holds:

$$t'_{i_1} \dots t'_{i_k} <_H^{\text{lex}} t_{i_1} \dots t_{i_k}$$

(Recall from Section 2.4 that $\leq_H^{\text{lex}}$ denotes the lexicographical order induced by $\leq_H$.)

Let us give a few examples in order to illustrate the notion of a termination witness. Firstly, the tuple (2) (or simply 2) is a termination witness for the recursive operator elem, because $l <_H \mathsf{cons}(y, l)$ (see Example 8.3.1). Secondly, if the induction lemma `minus-ind-lma` for the destructor minus (see Section 7.3.4) is contained in the data base, the tuple (1) is a termination witness for the operator div (see Section 7.3.1), for then we have $\mathsf{minus}(x, y) <_H x$ (regardless of the condition in the induction lemma). Thirdly, it is easy to see that *both* the tuples (1) and (2) are termination witnesses for the recursive operator minus as defined above. Finally, consider the following definition of the operator ack that axiomatizes Ackermann's function.

```
define operator  ack : Nat Nat --> Nat  with defining rules
ack-1:
  ack(0,y)       = s(y)
ack-2:
  ack(s(x),0)    = ack(x,s(0))
ack-3:
  ack(s(x),s(y)) = ack(x,ack(s(x),y)).
```

As can be expected (by those familiar with Ackermann's function), the tuple $(1, 2)$ is a termination witness for the recursive operator ack: from the definition of $<_H^{\text{lex}}$ it follows for the three recursive calls of ack that

- $(x, \mathsf{s}(0)) <_H^{\text{lex}} (\mathsf{s}(x), 0)$
- $(x, \mathsf{ack}(\mathsf{s}(x), y)) <_H^{\text{lex}} (\mathsf{s}(x), \mathsf{s}(y))$
- $(\mathsf{s}(x), y) <_H^{\text{lex}} (\mathsf{s}(x), \mathsf{s}(y))$

Observe that $(1, 2)$ is a *minimal* termination witness for ack, since neither the tuples (1) nor (2) are termination witnesses for ack.

Now in order to make its guess as to whether the recursive operator f is terminating, `analyze-operator` computes all the (minimal) termination witnesses for f. If none are found (which implies that no termination witness for f exists), then f is conjectured to be non-terminating and the analysis of f aborts. Otherwise, f is conjectured to be terminating.

Provided that f has been conjectured to be terminating, `analyze-operator` proceeds by computing the so-called *induction positions* of f. Intuitively seen, the induction positions of f are used to determine the induction variables during the construction of an inductive case analysis for a conjecture about f (see Section 8.3.3 for a more precise motivation). Formally, we define induction positions as follows. Assume that (1) $\alpha(f) = s_1 \ldots s_n s$; (2) $x_1, \ldots, x_n \in V^{\mathrm{C}}$ are the formal parameters of f; and (3) $\{\sigma_1, \ldots, \sigma_m, \sigma_{m+1}, \ldots, \sigma_{m+p}\}$ is the corresponding cover set of substitutions for the goal $\langle\, \mathrm{def}(f(x_1, \ldots, x_n))\ ;\ ()\rangle$. Then the integer $i \in \{1, \ldots, n\}$ is said to be an *induction position* of f if

(a) there is a $j \in \{1, \ldots, m+p\}$ such that $\sigma_j(x_i) \neq x_i$; or

(b) i occurs in any of the termination witnesses computed for f (given that f is recursive).

For instance, 1 is the only induction position of div and 2 the only induction position of elem, while 1 and 2 are the induction positions of minus and ack. Furthermore, suppose the following definition of the operator double is given:[22]

```
define operator  double: Nat --> Nat  with defining rules
double-1:
  double(x) = plus(x,x).
```

Apparently, this non-recursive operator does not have any induction positions ($m = 1$, $\sigma_1 = \mathrm{id}$ and $p = 0$) — which makes sense, since every call of double in a conjecture can (and should) be expanded without an inductive case analysis.

We can now (finally) describe the composition of the *definition scheme* of a defined operator f. As was already mentioned, the procedure `analyze-operator` computes a definition scheme of f in order to represent the results obtained throughout the analysis of the defining rules for f, and then creates an entry for it into the so-called operator table of the (proof control) data base. Essentially, the definition scheme of f is made up of the following objects:

- the formal parameters $x_1, \ldots, x_n \in V^{\mathrm{C}}$ of f
- a normalized representation of the defining rules for f (in terms of the formal parameters of f)
- the cover set of substitutions $\{\sigma_1, \ldots, \sigma_m, \sigma_{m+1}, \ldots, \sigma_{m+p}\}$ (for the goal $\langle\, \mathrm{def}(f(x_1, \ldots, x_n))\ ;\ ()\rangle$)
- a list of all the termination witnesses for f (if f is recursive)
- a list of all the induction positions of f (if f is conjectured to be terminating)

Once determined and stored at "definition-time", the definition scheme of f can then be retrieved from the data base at "proof-time", e.g. whenever an inductive case analysis needs to be constructed for a proof of a conjecture about f (see Section 8.3.3).

[22] see Example 2.2.1 for the defining rules for plus

The attentive reader may wonder why the definition scheme of f does not include, for each $i \in \{1, \ldots, m\}$, a special representation of the condition literals $\Delta_{i,1}, \ldots, \Delta_{i,k(i)}$ that occur in the "σ_i-rules", i.e. the defining rules for f whose left-hand side is the term $f(x_1\sigma_i, \ldots, x_n\sigma_i)$ (see above). For this purpose, we do have developed the notion of a *condition tree*, but for a reason to be explained in the following, condition trees are computed dynamically during the construction of inductive case analyses. Therefore, storing condition trees computed for $\sigma_1, \ldots, \sigma_m$ (as part of the definition scheme of f) in the operator table is superfluous. Nevertheless, we have to introduce condition trees at this point, because they are used in the final step to be taken by `analyze-operator`, namely throughout the generation of a conjecture about the domain of f (see below).

We mainly employ condition trees as a means of guiding (or "planning") expansions of calls of defined operators in goals. To be more precise, suppose a term of the form $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ occurring in a goal $\langle \Gamma\,;\, w\rangle$ — e.g. the term $\mathsf{elem}(x, \mathsf{cons}(y, l))$ in the goal at position 1.2 in the proof state tree `elem-def` from Example 8.3.1 — is to be expanded by applying the corresponding defining rules for f (the "σ_i-rules") — i.e. `elem-2 and elem-3` in the example. In general, this requires a series of carefully chosen applications of the inference rule Literal Addition for introducing the (equational) condition literals from $\Delta_{i,1}, \ldots, \Delta_{i,k(i)}$ that do not occur in Γ, before $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ can be "safely"[23] rewritten in as many of the resulting cases (i.e. subgoals) as possible. A condition tree for $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ and $\langle \Gamma\,;\, w\rangle$ is a tree-like structure that exactly describes (i) the applications of the inference rule Literal Addition needed to construct a case analysis for the expansion of $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ in Γ and (ii) which of the defining rules for f (i.e. "σ_i-rules") can be applied in the resulting cases. An inner node of a condition tree is labeled with a sequence of equational literals from $\Delta_{i,1}\mu, \ldots, \Delta_{i,k(i)}\mu$, whereas a leaf is labeled either with the name of a defining rule for f (a "σ_i-rule") or with "$\emptyset$". For instance, Figure 8.6 (a) depicts the condition tree computed for the expansion of the call $\mathsf{elem}(x, \mathsf{cons}(y, l))$ in the goal at position 1.2 in the proof state tree `elem-def` (i.e. $\mu = \mathrm{id}$).

A condition tree CT for $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ and $\langle \Gamma\,;\, w\rangle$ is intended to be "executed" (recursively) as follows: If CT consists of a leaf labeled with the name of a defining rule then $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ can be "safely" rewritten with that defining rule. If CT consists of a leaf labeled with "$\emptyset$" then there is no defining rule for expanding $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ in $\langle \Gamma\,;\, w\rangle$. Otherwise, the root of CT is an inner node ν labeled with some (equational) literals $\lambda_1, \ldots, \lambda_k$. These literals are the parameters of the application of the inference rule Literal Addition represented by ν. Moreover, each of the $k+1$ immediate subtrees of CT is a condition tree for the expansion of $f(x_1\sigma_i, \ldots, x_n\sigma_i)\mu$ in the subgoal $\langle \Gamma_j\,;\, w\rangle$, where $\langle \Gamma_1\,;\, w\rangle, \ldots, \langle \Gamma_{k+1}\,;\, w\rangle$ are the $k+1$ subgoals arising in the application of Literal Addition and $j \in \{1, \ldots, k+1\}$.

To illustrate this, we provide a few examples. Firstly, the root of the condition tree shown in Figure 8.6 (a) represents the application of Literal Addition ("`lit-add`") to the goal at position 1.2 in the proof state tree `elem-def` with the parameter $x = y$ (see Example 8.3.1). Besides, the condition tree also indicates that the resulting (sub-)

[23] see Example 8.3.1

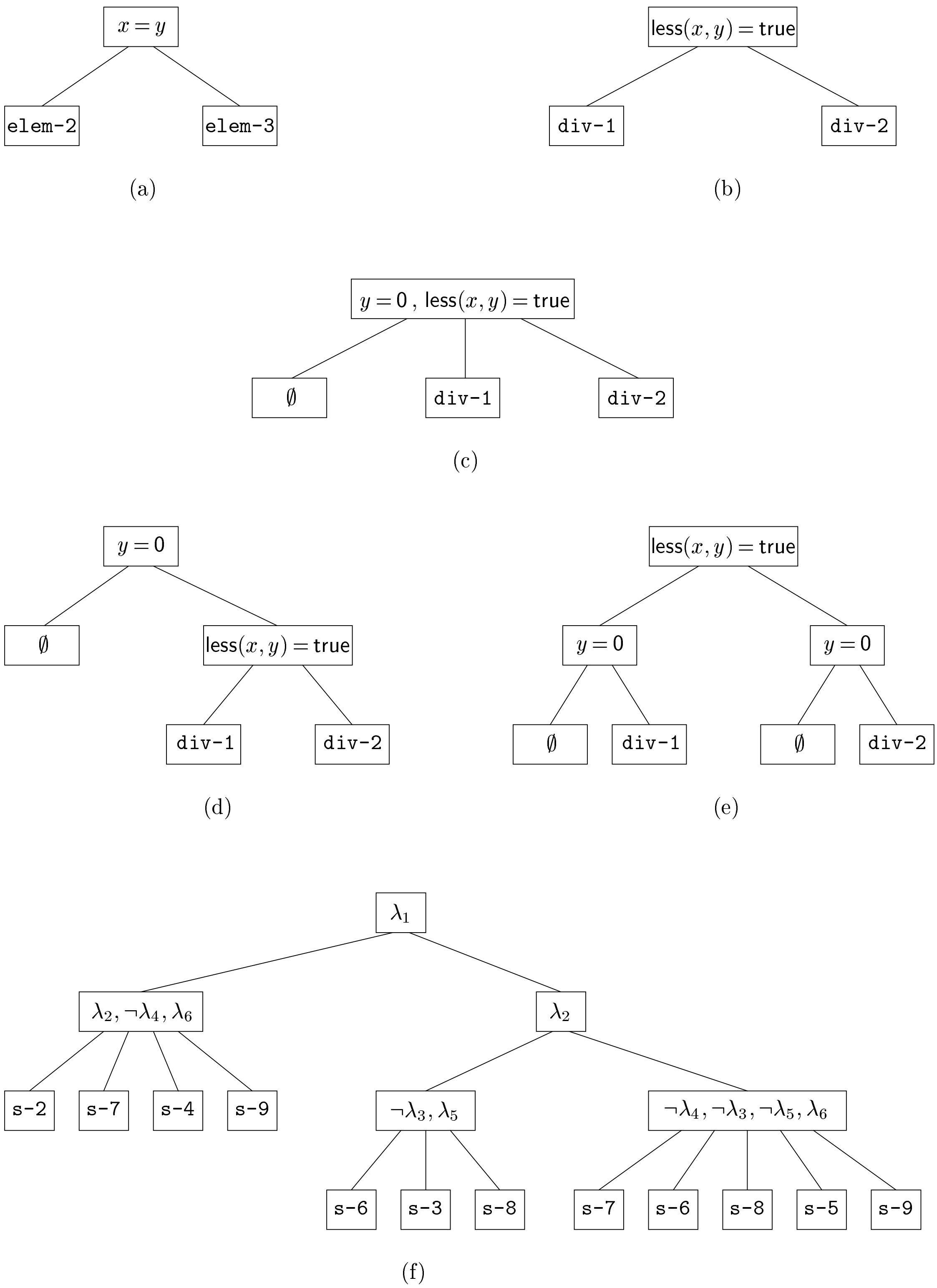


Figure 8.6: Examples of condition trees

goals (at positions $1.2.1^2$ and 1.2.1.2) can then be expanded with the defining rules `elem-2` and `elem-3`, respectively. Observe that the condition tree for expanding the call $\mathsf{elem}(x, \mathsf{nil})$ in the goal at position 1^2 is trivial: it merely consists of a leaf labeled with `elem-1`, since no condition literals need to be introduced for an expansion with this unconditional defining rule. Secondly, consider Figure 8.6 (b) which displays a condition tree for expanding the call $\mathsf{div}(x, y)$ in the goal $\langle\, \mathrm{def}(\mathsf{div}(x, y)) \vee y = \mathsf{0}\ ;\ x\rangle$ (i.e. the "domain lemma" for div). Since the condition literal $y \neq \mathsf{0}$ occurs in the (negated) context of the call, there is only one missing condition literal to be introduced in the case analysis represented by this condition tree, namely $\mathsf{less}(x, y) = \mathsf{true}$. The (different) condition tree computed for the expansion of $\mathsf{div}(x, y)$ in the goal $\langle\, \mathrm{def}(\mathsf{div}(x, y))\ ;\ x\rangle$ is displayed in Figure 8.6 (c). Essentially, this dependency on occurrences of condition literals in the context of the call to be expanded is the reason why condition trees are computed dynamically at "proof-time".

Mainly in order to be able to focus on more interesting issues in connection with our approach to proof control, we do not explain in this thesis how we compute condition trees that represent "good" case analyses for expansions of calls of defined operators with (conditional) defining rules. The procedure we have developed (and implemented in the QML module `condition-trees`) is based on relatively simple ideas, but technically complicated. One aspect of the problem may be illustrated with the condition trees displayed in Figures 8.6 (c) – (e). Each of them is a condition tree for the expansion of the call $\mathsf{div}(x, y)$ in the goal $\langle\, \mathrm{def}(\mathsf{div}(x, y))\ ;\ x\rangle$, but the three condition trees differ with regard to the size of the case analyses they represent (in terms of the number of applications of the inference rule Literal Addition and the number of the resulting cases, i.e. leaves). Due to the heuristic nature of our procedure, we cannot guarantee that it always computes an "optimal" condition tree, but so far the results have been rather satisfactory.

We conclude our presentation of condition trees at this point with another, "realistic" example. Section E.3.2 (in Appendix E) includes an axiomatization of the operator $\mathsf{search{-}tree{-}p}$ that can be used to decide whether or not a given binary tree is a search tree. The definition of $\mathsf{search{-}tree{-}p}$[24] comprises nine defining rules, eight of which (`s-tree-p-2` through `s-tree-p-9`) are "σ_2-rules", where $\sigma_1 = \{t \leftarrow \mathsf{empty{-}btree}\}$ and $\sigma_2 = \{t \leftarrow \mathsf{btree}(t_1, x, t_2)\}$ constitute a cover set of substitutions for the goal $\langle\, \mathrm{def}(\mathsf{search{-}tree{-}p}(t))\ ;\ ()\rangle$. The condition tree computed by our procedure for an expansion of the call $\mathsf{search{-}tree{-}p}(\mathsf{btree}(t_1, x, t_2))$ in the above goal is displayed in Figure 8.6 (f). Note that "s-i" stands for "s-tree-p-i", and the (equational) condition literals occurring in the "σ_2-rules" are abbreviated as follows:

λ_1:	$t_1 = \mathsf{empty{-}btree}$	λ_2:	$t_2 = \mathsf{empty{-}btree}$
λ_3:	$\mathsf{search{-}tree{-}p}(t_1) = \mathsf{true}$	λ_4:	$\mathsf{search{-}tree{-}p}(t_2) = \mathsf{true}$
λ_5:	$\mathsf{less}(\mathsf{rightmost{-}elem}(t_1), x) = \mathsf{true}$	λ_6:	$\mathsf{less}(x, \mathsf{leftmost{-}elem}(t_2)) = \mathsf{true}$

According to our experience, it is very difficult (if not practically impossible) to come

[24] Refer to Section E.3.1 for the definition of the sort for binary trees (including the constructors).

up with a "hand-made" case analysis for this expansion that is as compact as the one represented by the condition tree in Figure 8.6 (f).

Having introduced condition trees, we can now explain the final step in the analysis of a defined operator f by the procedure `analyze-operator`, namely the generation of a conjecture about the domain of f. In our formal framework for inductive theorem proving, statements on the domains of operators can be formulated with definedness atoms (see Section 3.1.1). These so-called *domain lemmas* are of practical significance, since they can usually be employed to directly prove the "definedness subgoals" arising from applications of applicative inference rules with non-empty definedness conditions (see Section 5.3 and Definition 5.3.1 in particular).

Let f be a defined operator with $\alpha(f) = s_1 \dots s_n s$. We say that a clause of the form $\mathrm{def}(f(x_1, \dots, x_n)), \Delta$ is a *domain lemma* for f, where $x_i \in V^{\mathrm{C}}_{s_i}$ for $i = 1, \dots, n$ and Δ is a clause. Note that in order to avoid awkward terminology, we do not require a domain lemma to be inductively valid (w.r.t. the current specification), which e.g. allows us to simply speak of domain lemmas in cases of falsely conjectured clauses of the above form.

Essentially, the heuristic method utilized by `analyze-operator` to conjecture a domain lemma for a *supposedly terminating* operator f is based on a test for the "completeness" of the case analysis underlying the definition of f. If the substitutions $\sigma_1, \dots, \sigma_m$ belonging to the definition scheme of f (see above) do not constitute a cover set of substitutions for the goal $\langle\, \mathrm{def}(f(x_1, \dots, x_n)\ ;\ ()\rangle$ then no estimate of the domain of f is made. Otherwise, a condition tree CT_i for expanding the call $f(x_1\sigma_i, \dots, x_n\sigma_i)$ in the goal $\langle\, \mathrm{def}(f(x_1\sigma_i, \dots, x_n\sigma_i)\ ;\ ()\rangle$ is computed for $i = 1, \dots, n$. If none of the condition trees $CT_1, \dots, CT_m$ have leaves labeled with "$\emptyset$" then f is assumed to be completely defined, and the conjectured (unconditional) domain lemma is $\mathrm{def}(f(x_1, \dots, x_n))$. In all other cases, `analyze-operator` does not make a guess at the domain of f. However, there is a specific (minor) exception to this: If (1) $m = 1$, (2) the root ν of CT_1 is labeled with literals $\lambda_1, \dots, \lambda_k$, and (3) the only leaf in CT_1 labeled with "$\emptyset$" is the first immediate successor of ν then the domain lemma $\mathrm{def}(f(x_1, \dots, x_n)) \vee \lambda_1$ is conjectured. As a consequence, `analyze-operator` determines the correct domain lemma $\mathrm{def}(\mathsf{div}(x, y)) \vee y = \mathsf{0}$ for the partially defined operator div, since the computed condition tree, which is depicted in Figure 8.6 (c), is of the required form.

Let us conclude the discussion of the procedure `analyze-operator` by presenting, as a short summary so to speak, the output of the command interpreter which is produced during the analysis of the operator elem (see Example 8.3.1). The command

```
call analyze-operator elem
```

yields the output

```
Analyzing the defining rules for operator elem ...

elem is recursive and conjectured to be terminating
```

```
Termination witness: (2)

Induction position:  2

Conjectured domain lemma:

{ def elem(x,l) }

Creating the PS-tree elem-def

The current PS-tree is elem-def

The current G-node in elem-def is root

Done
```

Observe that `analyze-operator` generates a proof state tree named `elem-def` for the conjectured domain lemma by calling the inference machine operation corresponding to the `prove` command (see Section 7.3.2).

Activating Lemmas

A further important procedure to be dealt with in connection with the (proof control) data base is the public procedure `activate-lemma`, which is also realized in the QML module `Database`. As sketched in the beginning of this subsection, the data base does not only consist of an operator table for the definition schemes of the analyzed defined operators, but it also contains a collection of the lemmas that have been *activated* so far. When activating a lemma (given in the form of a proof state tree), the procedure `activate-lemma` determines the type of the lemma and inserts it (or rather the name of the proof state tree) into the data base. A lemma can be either a domain lemma, an induction lemma, a rewrite lemma or a subsumption lemma (see below). Since the tactics realized in the QML module `Simplification` apply only activated lemmas (see Section 8.3.2), the essential effect of activating a lemma is to make it known to these tactics. Moreover, the extension of the relation $\leq_H$ for computing termination witnesses depends on the activation of induction lemmas (see above). Therefore, the user should (in most cases at least) activate a conjecture after it has been proved with QuodLibet. Although `activate-lemma` does not require a proof for the conjecture to be activated, the user is advised not to activate unproved conjectures, as this may lead to cyclic proof attempts.[25]

We have explained before what a domain lemma for a defined operator is. Let us now characterize the three remaining types of lemmas. Suppose f is a defined operator with $\alpha(f) = s_1 \dots s_n s$. Roughly speaking, *induction lemmas* may be seen as providing the relevant "termination arguments" in cases of definitions by *destructor* recursion or proofs by destructor induction (refer to Sections 4.3 and 7.3.4 for further explanations).

[25] due to the tactic `simplify-goal` (see Section 8.3.2)

Formally, an *induction lemma* for f is a clause of the form $f(x_1,\ldots,x_n) < x_j\,, \Delta$, where $x_i \in V^{\mathrm{C}}_{s_i}$ for $i = 1,\ldots,n$, $j \in \{1,\ldots,n\}$ and Δ is a clause. For instance, the clause $\mathsf{minus}(x,y) < x \ \vee\ \mathsf{less}(x,y) = \mathsf{true} \ \vee\ y = \mathtt{0}$ is an induction lemma for the destructor minus. For rewriting terms in inference steps with the inference rule Non-Inductive Rewriting, only so-called *rewrite lemmas* are taken into consideration by our simplification tactics, as will be explained in Section 8.3.2. We call a clause of the form $\Gamma,\ f(t_1,\ldots,t_n) = t\,, \Delta$ a *rewrite lemma* for f if $\mathrm{Var}(\Gamma, t, \Delta) \subseteq \mathrm{Var}(f(t_1,\ldots,t_n))$, where $t_i \in \mathcal{T}(sig, V)_{s_i}$ for $i = 1,\ldots,n$, and Γ and Δ are clauses. In addition to this, it is required that $\Gamma,\ f(t_1,\ldots,t_n) = t\,, \Delta$ is neither a domain nor an induction lemma for any operator. Observe that a clause can be a rewrite lemma for several operators: $\mathsf{less}(x,y) = \mathsf{true} \ \vee\ \mathsf{leq}(y,x) = \mathsf{true}$ is a rewrite lemma for both less and leq.[26] Finally, any clause (to be activated with `activate-lemma`) that is neither a domain lemma, an induction lemma nor a rewrite lemma for any operator is called a *subsumption lemma*. The only way a subsumption lemma Γ may contribute to a proof step constructed by one of our simplification tactics is the use of Γ in an application of the inference rule Non-Inductive Subsumption (see Section 8.3.2). Note that, as in case of domain lemmas, the above definitions do not require that induction, rewrite or subsumption "lemmas" be inductively valid (w.r.t. the current specification).

Displaying Information Stored in the Data Base

For the sake of completeness, let us mention the two remaining public procedures exported by the QML module `Database`, namely `display-operator-info` and `display-database`. The former of these procedures can be invoked to display (parts of) the definition scheme of a defined operator f and the lemmas for f activated so far, while the effect of the latter procedure is a listing of the contents of the whole data base.

8.3.2 Simplifying Clauses in Goals

Practical experience of inductive theorem proving shows that in general, the major part of the inferences made when constructing proofs by induction serve the purpose of *simplifying* formulas in goals. In the formal framework for inductive theorem proving presented in Part I, simplification steps may typically involve the recognition of simple tautologies, the removal of redundant literals, rewriting terms in clauses with defining rules or conjectures, or the subsumption with conjectures, where conjectures can be applied either non-inductively or inductively (refer to Sections 5.2 and 5.3 for the corresponding inference rules). Therefore, it is not surprising that as part of our effort to accomplish the desired (partial) automation of the proof control of QUODLIBET, we provide a variety of tactics for mechanically solving simplification tasks like the ones mentioned above. These tactics are realized in the QML module `Simplification`. Due to the considerable size of this module — it contains the QML definitions of twenty-one tactics, five procedures and fourteen functions — we can only sketch its eight public

[26] The definition of the operator leq ("less or equal") is given in Section E.1.1.

tactics in the following by roughly describing their behavior from the user's point of view. Observe that in Example 8.2.1, we explained the public tactic `prove-tautology` *in detail* (together with the auxiliary tactic `prove-taut-lit`), which was possible because `prove-tautology` is by far the simplest QML routine of this module.

Proving Simple Tautologies and Removing Redundant Literals

As was already said in Example 8.2.1, the public tactic `prove-tautology` can only be used to prove "tautologies", i.e. goals containing obviously inductively valid clauses. Therefore, the module `Simplification` provides a more general tactic named `cleanup` which is capable of both proving simple tautologies and removing "redundant" (i.e. obviously unsatisfiable) literals. Roughly speaking, the tactic `cleanup` tries to apply the inference rules from Sections 5.2.1 – 5.2.3 and the inference rule $\neq$-Unification (see Figure 5.6) to the given goal $\langle \Gamma\,;\, w\rangle$ (or to resulting subgoals of $\langle \Gamma\,;\, w\rangle$) as often as possible. For instance, when invoked for the goal

$$\langle\, \mathsf{false} = \mathsf{true} \;\vee\; \mathsf{s}(0) \neq \mathsf{s}(0) \;\vee\; \mathsf{less}(0, \mathsf{s}(\mathsf{s}(x))) = \mathsf{true}\ ;\ \ldots\rangle$$

`cleanup` applies in a first step the inference rule =-Removal, thus obtaining the goal

$$\langle\, \mathsf{s}(0) \neq \mathsf{s}(0) \;\vee\; \mathsf{less}(0, \mathsf{s}(\mathsf{s}(x))) = \mathsf{true}\ ;\ \ldots\rangle$$

It then reduces this goal in the next step to (the result) $\langle\, \mathsf{less}(0, \mathsf{s}(\mathsf{s}(x))) = \mathsf{true}\ ;\ \ldots\rangle$ by applying the inference rule $\neq$-Removal. For another example, consider the goal

$$\langle\, x = y \;\vee\; \mathsf{s}(x) \neq \mathsf{s}(y)\ ;\ ()\rangle$$

which represents the second of Peano's Postulates for the natural numbers. The tactic `cleanup` finds the following proof for this goal: Applying $\neq$-Unification to the second literal, `cleanup` reduces the conjecture to the goal $\langle\, y = y\ ;\ ()\rangle$, which it then proves with the inference rule =-Decomposition (see Figure 5.2).

Proving Definedness Subgoals

Recall from Section 5.3 that an application of an applicative inference rule may give rise to so-called *definedness subgoals* when the sequence of the definedness conditions $\mathrm{DefCond}(\mu, \Pi)$ of the involved matching substitution μ and the applied defining rule or conjecture Π is not empty (see Definition 5.3.1). For instance, rewriting the term $\mathsf{less}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z))), 0)$ with the defining rule $\mathsf{less}(x, 0) = \mathsf{false}$ in an application of the inference rule Non-Inductive Rewriting (normally) leads to a definedness subgoal of the form

$$\langle\, \mathrm{def}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z)))) \;\vee \ldots\ ;\ \ldots\rangle \tag{8.1}$$

due to the match $\mu = \{x \leftarrow \mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z)))\}$. In many cases, it is fairly easy to prove a definedness subgoal of the form $\langle\, \mathrm{def}(t) \vee \ldots\ ;\ w\rangle$ provided that for any defined operator f occurring in t an (unconditional) domain lemma for f has been activated before (see Section 8.3.1).

For automatically constructing proofs for definedness subgoals, the public tactic `prove-def-subgoal` is made available by the module `Simplification`. Let us briefly explain the overall idea as to how this tactic proceeds when it is called for a goal of the form $\langle\, \mathrm{def}(t) \;\vee \ldots\, ;\, w\rangle$. If t is a general variable then the call of `prove-def-subgoal` fails. If t is a constructor variable then the goal is proved directly with an application of the inference rule def-Decomposition (see Figure 5.3). Otherwise, t must be of the form $f(t_1, \ldots, t_n)$ for some $f \in F$. If f is a constructor then def-Decomposition is applied to the goal, and for each of the resulting subgoals the tactic `prove-def-subgoal` is called recursively. Otherwise, f is a defined operator, and a domain lemma for f is retrieved from the data base, which is then used in an application of the inference rule Non-Inductive Subsumption to the given goal. After that, the tactic `prove-def-subgoal` is called recursively for each of the definedness subgoals arising in this inference step.

To illustrate the tactic, we present the proof for the goal (8.1) that `prove-def-subgoal` constructs (assuming the domain lemma $\mathrm{def}(\mathsf{plus}(x,y))$ for plus has been activated). Since the leading function symbol of the term $\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y,z)))$ is a constructor, `prove-def-subgoal` applies the inference rule def-Decomposition to (8.1), which yields the (sub-) goal

$$\langle\, \mathrm{def}(\mathsf{plus}(x, \mathsf{plus}(y,z))) \;\vee\; \mathrm{def}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y,z)))) \;\vee \ldots\, ;\, \ldots\rangle \tag{8.2}$$

and `prove-def-subgoal` is recursively invoked for (8.2). The leading function symbol of the term $\mathsf{plus}(x, \mathsf{plus}(y,z))$ is the defined operator plus, and so `prove-def-subgoal` uses the domain lemma for plus in an application of the inference rule Non-Inductive Subsumption with the match $\mu = \{y \leftarrow \mathsf{plus}(y,z)\}$ to reduce (8.2) to the (sub-) goal

$$\langle\, \mathrm{def}(\mathsf{plus}(y,z)) \vee \mathrm{def}(\mathsf{plus}(x, \mathsf{plus}(y,z))) \vee \mathrm{def}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y,z)))) \vee \ldots\, ;\, \ldots\rangle \tag{8.3}$$

which is the definedness subgoal arising in this inference step. It is easy to see that `prove-def-subgoal`, now recursively invoked for (8.3), can prove (8.3) — and thus the original goal (8.1) — in one step with the domain lemma for plus in a further application of Non-Inductive Subsumption.

Proving Order Subgoals

As explained in Section 5.3.2, every inference step involving an inductive applicative inference rule gives rise to a so-called *order subgoal* that represents the order condition "guarding" the application of the induction hypothesis to the given goal. Typical examples of order subgoals are the two (schematic) goals

$$\langle\, (x, \mathsf{ack}(\mathsf{s}(x), y)) < (\mathsf{s}(x), \mathsf{s}(y)) \;\vee \ldots\, ;\, \ldots\rangle \tag{8.4}$$

$$\langle\, \mathsf{minus}(x,y) < x \;\vee \ldots \vee\; \mathsf{less}(x,y) = \mathsf{true} \;\vee \ldots \vee\; y = \mathsf{0}\; ;\; x\rangle \tag{8.5}$$

which may be generated throughout the construction of proofs for the two conjectures $\langle\, \mathsf{less}(y, \mathsf{ack}(x,y)) = \mathsf{true}\; ;\; (x,y)\rangle$ and $\langle\, \mathrm{def}(\mathsf{div}(x,y)) \;\vee\; y = \mathsf{0}\; ;\; x\rangle$, respectively.

The module `Simplification` provides a public tactic named `prove-order-subgoal` that succeeds in finding proofs for order subgoals of the form $\langle\, w_1 < w_2 \vee \ldots ;\, w\rangle$ in most practically relevant cases — given, however, that weight variables neither occur in w_1 nor in w_2[27] and that induction lemmas have been activated for the destructors in the current specification. Roughly speaking, `prove-order-subgoal` works as follows when invoked for an order subgoal of the form $\langle\, w_1 < w_2 \vee \ldots ;\, w\rangle$: If w_1 and w_2 are (non-trivial) tuple weights to be compared lexicographically then `prove-order-subgoal` attempts (i) to decompose them by means of the inference rules Tuple $<$-Reduction or Tuple $=$-Reduction (see Section 5.2.5) and (ii) to prove the resulting subgoal by a recursive invocation of the tactic. When both w_1 and w_2 are tuples of length one (i.e. terms), `prove-order-subgoal` tries to prove the order subgoal with an application of the inference rule $<$-Decomposition (see Figure 5.4). If the attempt to apply this inference rule fails and $w_1 < w_2$ matches the pattern $d(t_1, \ldots, t_l) < t_j$ for some defined operator d and $j \in \{1, \ldots, l\}$ then the tactic tries to retrieve a suitable (activated) induction lemma for d from the data base and use it to prove the order subgoal in a subsumption step with the inference rule Non-Inductive Subsumption.

Let us illustrate the tactic `prove-order-subgoal` by giving the proofs it constructs for the order subgoals (8.4) and (8.5). Firstly, since the two weights in the order atom in (8.4) are non-trivial tuples with different "leading" components (i.e. x and $\mathsf{s}(x)$), `prove-order-subgoal` applies the inference rule Tuple $<$-Reduction to (8.4), which yields the (sub-) goal

$$\langle\, x < \mathsf{s}(x) \;\vee\; (x, \mathsf{ack}(\mathsf{s}(x), y)) < (\mathsf{s}(x), \mathsf{s}(y)) \;\vee \ldots ;\, \ldots\rangle \tag{8.6}$$

Obviously, the recursive invocation of `prove-order-subgoal` for (8.6) leads to a proof for (8.6) — and thus for (8.4) —, as the tactic can prove (8.6) in one step with the inference rule $<$-Decomposition. Secondly, the attempt to apply $<$-Decomposition to the order atom in (8.5) fails. Therefore, `prove-order-subgoal` retrieves the (previously activated) induction lemma $\mathsf{minus}(x, y) < x \vee \mathsf{less}(x, y) = \mathsf{true} \vee y = \mathsf{0}$ for the destructor minus, which it then employs to prove (8.5) in a subsumption step with the inference rule Non-Inductive Subsumption.

Single Applications of Defining Rules and Conjectures

Each of the three public tactics to be introduced in the following is intended to perform essentially *one* simplification step that is given by an application of an *applicative* inference rule to a goal (see Section 5.3). Whereas the tactic `apply-axiom` allows terms to be rewritten with defining rules, the two other tactics, namely `apply-lemma` and `apply-ind-hypothesis`, can be used to apply conjectures in rewrite or subsumption steps either non-inductively or inductively.

Roughly speaking, the tactic `apply-axiom` is to determine the first (sub-) term in the given goal $\langle \Gamma ;\, w\rangle$ (using a left-to-right innermost strategy) to which any of the defining

[27] Refer to Section 7.3.2 for weight variables and the `set weight` command and to Section 8.3.4 for the `set-weights` procedure.

rules for the analyzed operators applies at top-level. If such a term t is found, the tactic rewrites t with the corresponding defining rule in an application of the inference rule Non-Inductive Rewriting to $\langle \Gamma\,;\, w\rangle$. In addition, `apply-axiom` invokes the tactic `prove-def-subgoal` for each of the definedness subgoals arising in the rewrite step, and it also applies (a restricted but more efficient variant of) the tactic `cleanup` to the subgoal containing the rewritten term. Note that in order for a defining rule $l = r \leftarrow \Delta$ to *apply* to t (at top-level), there must be a matching substitution μ such that, essentially, (1) $t = l\mu$ and (2) Γ contains $\overline{\Delta\mu}$.[28]

For instance, when invoked for the goal $\langle\, \mathsf{less}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z))), \mathsf{0}) = \mathsf{false}\ ;\ \ldots\rangle$, the tactic `apply-axiom` finds that the defining rule $\mathsf{less}(x, \mathsf{0}) = \mathsf{false}$ is applicable, and it employs this axiom to rewrite the term $\mathsf{less}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z))), \mathsf{0})$ in an application of Non-Inductive Rewriting. This leads to the two (sub-) goals

$$\langle\, \mathrm{def}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z)))) \,\vee\, \mathsf{less}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z))), \mathsf{0}) = \mathsf{false}\ ;\ \ldots\rangle \tag{8.7}$$

$$\langle\, \neg\mathrm{def}(\mathsf{s}(\mathsf{plus}(x, \mathsf{plus}(y, z)))) \,\vee\, \mathsf{false} = \mathsf{false}\ ;\ \ldots\rangle \tag{8.8}$$

As was shown above, (8.7) is proved by the tactic `prove-def-subgoal` which is called by `apply-axiom`. Besides, the variant of the tactic `cleanup` invoked by `apply-axiom` for the "rewritten" subgoal (8.8) is naturally capable of proving (8.8). Hence, the tactic `apply-axiom` constructs a complete proof for the original goal above.

The public tactic `apply-lemma` differs from `apply-axiom` in that it attempts to rewrite a term in the given goal $\langle \Gamma\,;\, w\rangle$ with (activated) *rewrite lemmas* (see Section 8.3.1) instead of defining rules. Furthermore, when none of the terms in Γ can be rewritten with the rewrite lemmas, the tactic checks for every *subsumption lemma* $\langle \Pi\,;\, \hat{w}\rangle$ (see Section 8.3.1) whether $\langle \Pi\,;\, \hat{w}\rangle$ subsumes $\langle \Gamma\,;\, w\rangle$.[29] If this is the case for a subsumption lemma $\langle \Pi\,;\, \hat{w}\rangle$ then `apply-lemma` applies the inference rule Non-Inductive Subsumption to $\langle \Gamma\,;\, w\rangle$ with $\langle \Pi\,;\, \hat{w}\rangle$. Again, the tactic `prove-def-subgoal` is invoked for each of the definedness subgoals that are generated in the subsumption step.

Observe that `apply-lemma` employs the "`lemma`"-variants of Non-Inductive Subsumption and Rewriting, while `apply-axiom` may only effect applications of the "`axiom`"-variant of Non-Inductive Rewriting (see Appendix C).

Essentially, the public tactic `apply-ind-hypothesis` is very similar to `apply-lemma`. However, there are three major differences. Firstly, for simplifying the given goal $\langle \Gamma\,;\, w\rangle$, the tactic `apply-ind-hypothesis` can (normally) only apply the goal at the root of the proof state tree to which $\langle \Gamma\,;\, w\rangle$ belongs, i.e. the *induction hypothesis* for $\langle \Gamma\,;\, w\rangle$ (instead of the activated rewrite and subsumption lemmas). Secondly, in the rewrite (or subsumption) step performed by `apply-ind-hypothesis`, the inference rule Inductive Rewriting (or Inductive Subsumption) is used. Thirdly, there is a further condition which the "candidate" conjecture $\langle \Pi\,;\, \hat{w}\rangle$ must meet in order to be applied to $\langle \Gamma\,;\, w\rangle$. This condition is based on a heuristic analysis of the weights w and $\hat{w}$

[28] i.e. t can be "safely" rewritten with $l = r \leftarrow \Delta$ (see Example 8.3.1)

[29] (roughly) in the sense defined in Section 5.3.1

and is to ensure that an obviously futile application of $\langle \Pi\,;\,\hat{w}\rangle$ to $\langle \Gamma\,;\,w\rangle$ as induction hypothesis is prevented. For example, this condition is not fulfilled if $\hat{w}\mu = w$ (where μ is the involved match) — in this case, an order subgoal of the form $\langle\, w < w \vee \ldots\,;\,\ldots\rangle$ would be generated.

A General Purpose Simplification Tactic

Unquestionably, `simplify-goal` as the final tactic to be presented in this subsection is the practically most relevant of the public tactics made available by the QML module `Simplification`. This tactic virtually integrates the previously introduced "special purpose" simplification tactics[30] into an effective "general purpose" simplification tactic and is intended to be employed by our strategy tactics (see Section 8.3.4) and by the user. As the example proof constructions presented in Appendix E imply, the tactic `simplify-goal` is particularly useful during the interactive and *semi-automated* construction of proofs for non-trivial inductive theorems. By automatically performing relatively simple but tedious inference steps, `simplify-goal` helps the user concentrate on crucial issues throughout the proof construction, such as finding missing lemmas. Besides, our experience shows that, when invoked for a goal $\langle \Gamma\,;\,w\rangle$ that can be proved without an inductive case analysis and without applications of induction hypotheses, `simplify-goal` often succeeds in constructing a proof for $\langle \Gamma\,;\,w\rangle$.

We briefly explain the overall idea as to how the tactic `simplify-goal` works when called for a goal $\langle \Gamma\,;\,w\rangle$. In the first step, the tactic `cleanup` is invoked for $\langle \Gamma\,;\,w\rangle$. If this call proves $\langle \Gamma\,;\,w\rangle$ then `simplify-goal` terminates successfully. Otherwise, a (LIFO) list OPEN for open (sub-) goals is initialized with the subgoals generated by `cleanup` (or with $\langle \Gamma\,;\,w\rangle$ itself). In the ensuing steps, the list OPEN is processed as long as it still contains elements. The tactic always processes the first goal $\langle \Gamma'\,;\,w'\rangle$ of OPEN. After removing $\langle \Gamma'\,;\,w'\rangle$ from OPEN, `simplify-goal` applies (a variant of) the tactic `apply-axiom` to $\langle \Gamma'\,;\,w'\rangle$. If `apply-axiom` simplifies $\langle \Gamma'\,;\,w'\rangle$ then the resulting subgoal is inserted into OPEN (unless `apply-axiom` proved $\langle \Gamma'\,;\,w'\rangle$), and `simplify-goal` processes the next open goal in OPEN. Otherwise, the tactic invokes (a variant of) `apply-lemma` for $\langle \Gamma'\,;\,w'\rangle$. If `apply-lemma` reduces $\langle \Gamma'\,;\,w'\rangle$ to a subgoal $\langle \Gamma''\,;\,w''\rangle$ in a *rewrite* step then $\langle \Gamma''\,;\,w''\rangle$ is added to OPEN, and the next open goal node is processed. Otherwise, OPEN is not changed, and `simplify-goal` processes the next element of OPEN. The tactic terminates when OPEN has become empty.

For an example of a proof constructed by the tactic `simplify-goal` consider Figure 8.7. It depicts a proof (state tree) which shows how `simplify-goal` proves the conjecture

$$\langle\, \mathsf{times}(x,y) = \mathsf{double}(x) \;\vee\; y \neq \mathsf{s}(\mathsf{s}(0))\;;\;()\rangle$$

in a series of (i) one application of the inference rule $\neq$-Unification, (ii) four consecutive rewrite steps with the "`axiom`"-variant of Non-Inductive Rewriting, (iii) one rewrite step with the "`lemma`"-variant of Non-Inductive Rewriting, and finally (iv) one application of

[30]with the exception of `apply-ind-hypothesis`

=-Decomposition. Note that for the construction of this proof by `simplify-goal`, it is necessary that all of the involved defined operators have been analyzed and that the rewrite lemma $\mathsf{plus}(0, y) = y$ [31] has been activated. The defining rules applied in the proof are:

```
times-1:   times(x,0) = 0
times-2:   times(x,s(y)) = plus(times(x,y),x)

double-1:  double(x) = plus(x,x)
```

8.3.3 Constructing Inductive Case Analyses

As discussed in Section 8.3.1 and illustrated in Example 8.3.1, "promising" inductive case analyses for proofs of inductive theorems can often be obtained by means of the well-known *induction heuristic*. Roughly speaking, an inductive case analysis for the proof of a conjecture Γ is to generate the cases necessary for *expanding* certain, carefully selected calls of (recursive) defined operators occurring in Γ with the defining rules for these operators. Therefore, it is heuristically plausible that the construction of an inductive case analysis is guided by the case analyses used in the definitions of the selected defined operators. In Section 8.3.1, we described in detail how we analyze the axiomatization of a defined operator f and summarize the obtained information in the *definition scheme* of f as a representation of the case analysis used in the definition of f. In the following, we will explain how the public tactic `ind-case-analysis`, which the QML module `Inductive-Case-Analyses` makes available, employs these definition schemes when (1) constructing an inductive case analysis for the proof of a conjecture about *several* operators *and* (2) expanding the selected calls.

Speaking in simplified terms, the tactic `ind-case-analysis` proceeds as follows when applied to a goal $\langle \Gamma\,;\, w\rangle$.

In the first step, `ind-case-analysis` determines the so-called *expandable* calls of the defined operators occurring in Γ and stores them in the list E-CALLS. Roughly stated, an expandable call is "general" enough to be expanded with a case analysis, and thus it suggests an induction. Suppose f is a defined operator with $\alpha(f) = s_1 \dots s_n s$ that is conjectured to be terminating. Let $i_1, \dots, i_k \subset \{1, \dots, n\}$ be the induction positions of f, where $k > 0$ (see Section 8.3.1). A call of f, i.e. a term of the form $f(t_1, \dots, t_n)$ is said to be *expandable* if $\{t_{i_1}, \dots, t_{i_k}\}$ is a set of k distinct constructor variables. For instance, the calls $\mathsf{app}(l, \mathsf{nil})$ and $\mathsf{app}(l, l)$ are expandable (1 is the only induction position of app[32]), whereas the calls $\mathsf{less}(0, y)$ or $\mathsf{less}(x, x)$ are not (both 1 and 2 are the induction positions of less).

If there is no occurrence of an expandable call in $\langle \Gamma\,;\, w\rangle$ (i.e. the list E-CALLS is empty) then `ind-case-analysis` computes a "standard" cover set of substitutions for

[31] which has the name "`plus-0-y`"
[32] see Section E.2.2 for the definition of app

```
+-G-node root  "times-double"
  |  { times(x,y) = double(x),
  |    y =/= s(s(0)) } ;
  |  ()
  |
  +-I-node [1]
    | < =/=-unif 2 >
    |
    +-G-node [1^2]
      |  { times(x,s(s(0))) = double(x) } ;
      |  ()
      |
      +-I-node [1^3]
        | < axiom-rewrite 1 [1] times-2 1 [y <-- s(0)] >
        |
        +-G-node [1^4]
          |  { plus(times(x,s(0)),x) = double(x) } ;
          |  ()
          |
          +-I-node [1^5]
            | < axiom-rewrite 1 [1^2] times-2 1 [y <-- 0] >
            |
            +-G-node [1^6]
              |  { plus(plus(times(x,0),x),x) = double(x) } ;
              |  ()
              |
              +-I-node [1^7]
                | < axiom-rewrite 1 [1^3] times-1 1 [ ] >
                |
                +-G-node [1^8]
                  |  { plus(plus(0,x),x) = double(x) } ;
                  |  ()
                  |
                  +-I-node [1^9]
                    | < axiom-rewrite 1 [2] double-1 1 [ ] >
                    |
                    +-G-node [1^10]
                      |  { plus(plus(0,x),x) = plus(x,x) } ;
                      |  ()
                      |
                      +-I-node [1^11]
                        | < lemma-rewrite 1 [1^2] plus-0-y 1 [y <-- x] >
                        |
                        +-G-node [1^12]
                          |  { plus(x,x) = plus(x,x) } ;
                          |  ()
                          |
                          +-I-node [1^13]
                              < =-decomp 1 >
```

Figure 8.7: A proof (state tree) constructed by the tactic `simplify-goal`

all of the constructor variables occurring in $\langle \Gamma\,;\,w\rangle$. Subsequently, the tactic applies the inference rule Substitution Addition to $\langle \Gamma\,;\,w\rangle$ using the computed cover set of substitutions as the parameter. Moreover, after calling the tactic `prove-tautology` for each of the resulting subgoals, `ind-case-analysis` terminates (successfully). For example, there is no expandable call in the goal

$$\langle\, \mathsf{less}(\mathsf{0}, y) = \mathsf{true} \,\lor\, y = \mathsf{0} \;;\; y \rangle$$

Hence, `ind-case-analysis` reduces the goal by applying Substitution Addition with the "standard" cover set of substitutions for y, namely $\{\{y \leftarrow \mathsf{0}\}, \{y \leftarrow \mathsf{s}(y)\}\}$. This inference step yields the two "cases"

$$\langle\, \mathsf{less}(\mathsf{0}, \mathsf{0}) = \mathsf{true} \,\lor\, \mathsf{0} = \mathsf{0} \;;\; \mathsf{0} \rangle$$

$$\langle\, \mathsf{less}(\mathsf{0}, \mathsf{s}(y)) = \mathsf{true} \,\lor\, \mathsf{s}(y) = \mathsf{0} \;;\; \mathsf{s}(y) \rangle$$

the first of which `ind-case-analysis` proves with an invocation of `prove-tautology`.

Provided that the list E-CALLS of expandable calls occurring in $\langle \Gamma\,;\,w\rangle$ is not empty, `ind-case-analysis` deletes the *"obstructing"* calls from E-CALLS in the next step. Inspired by the notion of a *flawed* induction scheme by Boyer and Moore (1979), we heuristically eliminate "obstructing" expandable calls, because expansions of "obstructing" calls tend to cause redundant inference steps, as is suggested by numerous example proofs. Consider e.g. the conjecture

$$\langle\, \mathsf{elem}(x, \mathsf{app}(l_1, l_2)) \neq \mathsf{true} \,\lor\, \mathsf{elem}(x, l_1) = \mathsf{true} \,\lor\, \mathsf{elem}(x, l_2) = \mathsf{true} \;;\; \ldots \rangle \tag{8.9}$$

Apparently, $\mathsf{app}(l_1, l_2)$, $\mathsf{elem}(x, l_1)$ and $\mathsf{elem}(x, l_2)$ are the expandable calls in (8.9), but $\mathsf{elem}(x, l_2)$ "obstructs" $\mathsf{app}(l_1, l_2)$ in the sense that l_2, which occurs at an induction position in $\mathsf{elem}(x, l_2)$, is a sub-term of an argument in the call $\mathsf{app}(l_1, l_2)$ which is not at an induction position. It is easy to see that for a proof of (8.9), only $\mathsf{app}(l_1, l_2)$ and $\mathsf{elem}(x, l_1)$ need to be expanded, which yields an induction on l_1, whereas the additional expansion of $\mathsf{elem}(x, l_2)$ would lead to an unnecessary and "more expensive" induction on l_1 and l_2.

Formally, we define "obstruction" as follows. Let g be another defined operator with $\alpha(g) = \bar{s}_1 \ldots \bar{s}_l \bar{s}$. Moreover, assume that g is conjectured to be terminating and that $j_1, \ldots, j_q \subset \{1, \ldots, l\}$ are the induction positions of g, where $q > 0$. We say that the call $f(t_1, \ldots, t_n)$ *obstructs* the call $g(\bar{t}_1, \ldots, \bar{t}_l)$ if there is an induction position i of f and a $j \in \{1, \ldots, l\} \backslash \{j_1, \ldots, j_q\}$ such that t_i is a sub-term of $\bar{t}_j$.

As said above, the tactic `ind-case-analysis` deletes each call from E-CALLS that obstructs any other expandable call in $\langle \Gamma\,;\,w\rangle$. However, if E-CALLS turns out to be empty after this elimination process[33] then the list consisting of the first expandable call in $\langle \Gamma\,;\,w\rangle$ is assigned to E-CALLS.

At this point, E-CALLS is not empty and contains those expandable calls in $\langle \Gamma\,;\,w\rangle$ that will actually be expanded. For this purpose, `ind-case-analysis` first computes

[33] as in the case of the goal $\langle\, \mathsf{plus}(x, y) = \mathsf{plus}(y, x) \;;\; \ldots \rangle$

a cover set of substitutions for $\langle \Gamma\,;\,w\rangle$. The tactic does that by *merging* the cover sets of substitutions that are associated with the calls in E-CALLS and stored in the data base as components of the corresponding definition schemes (see Section 8.3.1). Instead of describing the (straightforward) procedure we developed for merging cover sets of substitutions, we give the following examples in order to convey an intuitive impression of how cover sets of substitutions are merged.

- The result of merging a cover set of substitutions $\{\sigma_1,\ldots,\sigma_m\}$ with itself is $\{\sigma_1,\ldots,\sigma_m\}$.
- Let $\sigma_1=\{x\leftarrow \mathsf{0}\}$, $\sigma_2=\{x\leftarrow \mathsf{s}(x)\}$, $\sigma_3=\{x\leftarrow \mathsf{s}(\mathsf{0})\}$ and $\sigma_4=\{x\leftarrow \mathsf{s}(\mathsf{s}(x))\}$. Then merging $\{\sigma_1,\sigma_2\}$ and $\{\sigma_1,\sigma_3,\sigma_4\}$ yields $\{\sigma_1,\sigma_3,\sigma_4\}$.
- Let $\sigma_1=\{x\leftarrow \mathsf{0}, y\leftarrow \mathsf{s}(y)\}$, $\sigma_2=\{y\leftarrow \mathsf{0}\}$, $\sigma_3=\{x\leftarrow \mathsf{s}(x), y\leftarrow \mathsf{s}(y)\}$, and let $\sigma'_1=\{y\leftarrow \mathsf{0}, z\leftarrow \mathsf{s}(z)\}$, $\sigma'_2=\{z\leftarrow \mathsf{0}\}$, $\sigma'_3=\{y\leftarrow \mathsf{s}(y), z\leftarrow \mathsf{s}(z)\}$.

  Then the result of merging $\{\sigma_1,\sigma_2,\sigma_3\}$ and $\{\sigma'_1,\sigma'_2,\sigma'_3\}$ is $\{\sigma''_1,\ldots,\sigma''_6\}$, where

$$
\begin{array}{ll}
\sigma''_1=\{x\leftarrow \mathsf{0}, y\leftarrow \mathsf{s}(y), z\leftarrow \mathsf{0}\} & \sigma''_2=\{x\leftarrow \mathsf{0}, y\leftarrow \mathsf{s}(y), z\leftarrow \mathsf{s}(z)\}\\
\sigma''_3=\{y\leftarrow \mathsf{0}, z\leftarrow \mathsf{s}(z)\} & \sigma''_4=\{y\leftarrow \mathsf{0}, z\leftarrow \mathsf{0}\}\\
\sigma''_5=\{x\leftarrow \mathsf{s}(x), y\leftarrow \mathsf{s}(y), z\leftarrow \mathsf{0}\} & \sigma''_6=\{x\leftarrow \mathsf{s}(x), y\leftarrow \mathsf{s}(y), z\leftarrow \mathsf{s}(z)\}
\end{array}
$$

Having obtained a cover set of substitutions $\{\sigma_1,\ldots,\sigma_q\}$ for $\langle \Gamma\,;\,w\rangle$ as the result of (successively) merging the cover sets of substitutions for the calls in E-CALLS, the tactic `ind-case-analysis` applies the inference rule Substitution Addition to $\langle \Gamma\,;\,w\rangle$ with $\{\sigma_1,\ldots,\sigma_q\}$ as the parameter. This yields the subgoals $\langle \Gamma\sigma_1\,;\,w\sigma_1\rangle,\ldots,\langle \Gamma\sigma_q\,;\,w\sigma_q\rangle$, for each of which the tactic `prove-tautology` is invoked.

In each of the ensuing q steps (for $i=1,\ldots,q$), `ind-case-analysis` expands the calls included in E-CALLS, the first call in the goal $\langle \Gamma\sigma_i\,;\,w\sigma_i\rangle$[34] and the remaining ones in the resulting subgoals. Suppose $f(t_1,\ldots,t_n)$ is the first call in E-CALLS. In order to have a "plan" for the expansion of $f(t_1,\ldots,t_n)\sigma_i$ in $\langle \Gamma\sigma_i\,;\,w\sigma_i\rangle$, `ind-case-analysis` computes a condition tree for $f(t_1,\ldots,t_n)\sigma_i$ and $\langle \Gamma\sigma_i\,;\,w\sigma_i\rangle$ (see Section 8.3.1). Then the tactic "executes" this condition tree (as explained in Section 8.3.1), which gives rise to a series of applications of the inference rules Literal Addition and Non-Inductive Rewriting. In addition, `ind-case-analysis` invokes (i) the tactic `prove-def-subgoal` for each of the definedness subgoals generated in the rewrite steps with defining rules for f and (ii) the tactic `prove-taut-lit` (see Figure 8.3) for each of the subgoals containing the rewritten call of f. After this expansion of the first call in E-CALLS, `ind-case-analysis` expands, in the way just described, the remaining calls in E-CALLS in each of the open subgoals created during the expansion of the first call.

Subsequent to this description of `ind-case-analysis`, let us illustrate the way in which the tactic proceeds when applied to the example conjecture (8.9). In the notation used by the QUODLIBET command interpreter, (8.9) is presented as follows:[35]

[34] unless the call of `prove-tautology` has proved $\langle \Gamma\sigma_i\,;\,w\sigma_i\rangle$

[35] Apparently, the name of the proof state tree created for (8.9) is `elem-app-3`.

```
G-node root  "elem-app-3"
 { elem(x,append(l1,l2)) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(l1,x,l2)
```

As mentioned above, there are three expandable calls in the conjecture. However, the tactic `ind-case-analysis` decides to expand only the calls $\mathsf{app}(l_1, l_2)$ and $\mathsf{elem}(x, l_1)$, because the third call $\mathsf{elem}(x, l_2)$ obstructs $\mathsf{app}(l_1, l_2)$. The result of merging the cover sets of substitutions associated with the two calls to be expanded is the cover set of substitutions $\{\sigma_1, \sigma_2\}$, where $\sigma_1 = \{l_1 \leftarrow \mathsf{nil}\}$ and $\sigma_2 = \{l_1 \leftarrow \mathsf{cons}(y, l_1)\}$. Therefore, the ensuing application of the inference rule Substitution Addition to the conjecture with $\{\sigma_1, \sigma_2\}$ as the parameter yields the two subgoals

```
G-node [1^2]
 { elem(x,append(nil,l2)) =/= true,
   elem(x,nil) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(nil,x,l2)

G-node [1:2]
 { elem(x,append(cons(y,l1),l2)) =/= true,
   elem(x,cons(y,l1)) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

In the next step, `ind-case-analysis` expands the first call, namely $\mathsf{app}(l_1, l_2)$, in the goal at position 1^2. The condition tree for the expansion of $\mathsf{app}(\mathsf{nil}, l_2)$ in this goal is trivial, it consists of one leaf labeled with the name `app-1` of the corresponding defining rule for app. Hence, the call $\mathsf{app}(\mathsf{nil}, l_2)$ is rewritten with `app-1` in an application of the inference rule Non-Inductive Rewriting, which gives rise to the subgoal

```
G-node [1^4]
 { elem(x,l2) =/= true,
   elem(x,nil) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(nil,x,l2)
```

Since the invocation of the tactic `prove-taut-lit` for the (rewritten) goal at position 1^4 proves this goal,[36] `ind-case-analysis` continues with the expansion of the first call in the goal at position 1.2. Again, the condition tree for expanding $\mathsf{app}(\mathsf{cons}(y, l_1), l_2)$ in this goal consists of one leaf, this time for the defining rule `app-2`. In the subsequent rewrite step with `app-2`, the goal at position 1.2 is reduced to the subgoal

[36] with the inference rule Complementary Literals (see Figure 5.1)

```
G-node [1:2:1^2]
 { elem(x,cons(y,append(l1,l2))) =/= true,
   elem(x,cons(y,l1)) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

Now the remaining expandable call, namely $\mathsf{elem}(x, l_1)$, is dealt with. Figure 8.6 (a) depicts the condition tree that `ind-case-analysis` computes for expanding the call $\mathsf{elem}(x, \mathsf{cons}(y, l_1))$ in the goal at position $1.2.1^2$. In "executing" this condition tree, the tactic first applies the inference rule Literal Addition to this goal (in order to introduce the missing condition literal $x = y$). This yields the two subgoals

```
G-node [1:2:1^4]
 { x =/= y,
   elem(x,cons(y,append(l1,l2))) =/= true,
   elem(x,cons(y,l1)) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)

G-node [1:2:1^3:2]
 { x = y,
   elem(x,cons(y,append(l1,l2))) =/= true,
   elem(x,cons(y,l1)) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

The leaves of the condition tree indicate how these two subgoals are to be rewritten: the goal at $1.2.1^4$ with the defining rule `elem-2` and the goal at $1.2.1^3.2$ with the defining rule `elem-3`. The subgoals generated in these *two* applications of the inference rule Non-Inductive Rewriting are

```
G-node [1:2:1^6]
 { x =/= y,
   elem(x,cons(y,append(l1,l2))) =/= true,
   true = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)

G-node [1:2:1^3:2:1^2]
 { x = y,
   elem(x,cons(y,append(l1,l2))) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

Since `ind-case-analysis` proves the goal at position $1.2.1^6$ with a call of the tactic `prove-taut-lit`, there is only one open goal still to be proved — this goal may be considered the "result" of invoking `ind-case-analysis` for the conjecture (8.9).

It should be noted that due to its heuristic nature, the tactic `ind-case-analysis` does not always make the "right" choice when it has to decide which of the expandable calls in the given conjecture should actually be expanded. For instance, when applied to the goal[37]

$$\langle\, \mathsf{less}(x, \mathsf{plus}(y, z)) = \mathsf{true} \vee \mathsf{less}(x, y) \neq \mathsf{true}\ ;\ \ldots \rangle$$

`ind-case-analysis` eliminates the expandable call $\mathsf{less}(x, y)$, because it obstructs the other expandable call $\mathsf{plus}(y, z)$. However, the conjecture should be proved by induction on x and y (as suggested by $\mathsf{less}(x, y)$) instead of z (as suggested by $\mathsf{plus}(y, z)$). Mainly in order to allow the user to deal with situations like these more conveniently, the QML module `Inductive-Case-Analyses` provides a second public tactic named `expand-operator`. As its name implies, the tactic `expand-operator` constructs the inductive case analysis that is suggested by *one* expandable call which the user supplies as a parameter. Roughly speaking, the tactic `expand-operator` works as follows when invoked for a goal $\langle \Gamma\, ;\, w \rangle$ and an expandable call $f(t_1, \ldots, t_n)$ occurring in $\langle \Gamma\, ;\, w \rangle$. In the first step, it applies the inference rule Substitution Addition with the cover set of substitutions $\sigma_1, \ldots, \sigma_{m+p}$ contained in the definition scheme of f. Then it expands, for $i = 1, \ldots, m + p$, the call $f(t_1, \ldots, t_n)\sigma_i$ in the goal $\langle \Gamma\sigma_i\, ;\, w\sigma_i \rangle$ by computing and "executing" the corresponding condition tree.

8.3.4 Inductive Proof Strategies

Having described a variety of simplification tactics and the tactic `ind-case-analysis` for the heuristic generation of inductive case analyses in the previous subsections, we can now present the public tactic `standard-strategy`, which integrates the tactics discussed so far into a full-scale inductive proof strategy (or proof procedure) and may be regarded as one of the main achievements of this thesis. As the example proof constructions given in Appendix E show, the tactic `standard-strategy` is capable of mechanically constructing complete proofs for many (simpler) inductive theorems, and thus it realizes the automation of the proof control of QUODLIBET as required in Section 7.1 to a reasonable extent. In addition to `standard-strategy`, the module `Proof-Strategies` provides (i) another public tactic named `restricted-strategy`, which is a restricted variant of `standard-strategy`, as well as (ii) the public procedure `set-weights` (see Figure 8.1). Since these two strategy tactics make use of `set-weights`, we will briefly describe this procedure at first, before dealing with the two tactics exported by `Proof-Strategies`.

Setting Weight Variables

Recall from Section 7.3.2 that whenever a proof state tree T is created for a conjecture Γ, a *weight variable* $\mathtt{w\_}T(x_1, \ldots, x_l)$ is generated for the weights of the goals in T, where $x_1, \ldots, x_l$ are the constructor variables occurring in Γ. Moreover, no inference rule nor the tactic `prove-order-subgoal` can be applied to an order subgoal in T as long as

[37] see the conjecture `less-x-plus-y-z` in Section E.1.2

it still contains weight variables. Therefore, the weight variable $\mathtt{w\_}T(x_1, \ldots, x_n)$ must be set with a suitable concrete weight, before the proof construction represented by T can be completed. For this purpose, we provide the public procedure `set-weights`.

Speaking in simplified terms, the procedure `set-weights` can be explained as follows. Suppose `set-weights` is applied to the proof state tree T for the conjecture $\langle \Gamma\,;\, w\rangle$, where $x_1, \ldots, x_l$ are the constructor variables occurring in Γ. The procedure is based on the assumptions (1) that all of the order subgoals to arise in the construction of T are available at the point when `set-weights` is invoked for T and (2) that each of these order subgoals is of the form

$$\langle\, \mathtt{w\_}T(t'_1, \ldots, t'_l) < \mathtt{w\_}T(t_1, \ldots, t_l)\,, \Delta\,;\, \mathtt{w\_}T(t_1, \ldots, t_l)\rangle$$

Essentially, `set-weights` attempts to compute a tuple $i_1 \ldots i_k \subseteq \{1, \ldots, l\}^*$ of k distinct integers such that for each order subgoal of the above form in T the following condition holds:[38]

$$t'_{i_1} \ldots\ t'_{i_k}\ <^{\mathrm{lex}}_H\ t_{i_1} \ldots\ t_{i_k}$$

If the procedure finds a tuple $i_1, \ldots, i_k$ with the required property, it assigns the weight $(x_{i_1}, \ldots, x_{i_k})$ to the weight variable $\mathtt{w\_}T(x_1, \ldots, x_l)$ with a call of the inference machine operation (or library procedure) that corresponds to the `set weight` command. As was explained in Section 7.3.2, the result of this is that each occurrence of the weight variable $\mathtt{w\_}T$ in the current proof state graph is consistently replaced with the computed weight. In case that no such tuple can be determined, the call of `set-weights` fails.

For instance, the only order subgoal arising in the construction of the proof state tree `div-def` for the domain lemma for `div` is of the form

```
G-node [1:2:1:2:1^3:2]
 { w_div-def(minus(x,y),y) < w_div-def(x,y),
   ...
   y = 0 } ;
 w_div-def(x,y)
```

(see Section 7.3.4). Provided that the data base contains the required definition schemes and the *activated* domain lemma for `minus`, `set-weights` finds the tuple (1), since the inequality $\mathsf{minus}(x, y) <_H x$ holds. Therefore, the procedure assigns the weight x to the weight variable `w_div-def`, and the above order subgoal is changed to

```
G-node [1:2:1:2:1^3:2]
 { minus(x,y) < x,
   ...
   y = 0 } ;
 x
```

A further example of an invocation of the procedure `set-weights` will be given below. Note that, due to the definition of $\leq_H$ and our practical experience (see Appendix E), it is justifiable to maintain that in most of the practically relevant cases, `set-weights` determines suitable weights that lead to provable order subgoals.

[38] Refer to Section 8.3.1 for the definition of $<^{\mathrm{lex}}_H$.

The Strategy Tactic `standard-strategy`

We now present the tactic `standard-strategy` which is the most effective of the proof control routines we have developed so far. In what follows, we are going to explain — partly in strongly simplified terms — how this strategy tactic combines the previously introduced tactics to form a comprehensive inductive proof procedure.

The tactic `standard-strategy` can be invoked for any goal $\langle \Gamma\, ;\, w \rangle$ which labels the root of a proof state tree T. In the first step, `standard-strategy` applies the "general purpose" simplification tactic `simplify-goal` (see Section 8.3.2) to $\langle \Gamma\, ;\, w \rangle$, because conjectures to be proved by induction should be as "simplified" as possible (see e.g. Boyer & Moore, 1979).

Let us first deal with the case that $\langle \Gamma\, ;\, w \rangle$ is simplified, i.e. the call of `simplify-goal` fails. After the invocation of the tactic `ind-case-analysis` for $\langle \Gamma\, ;\, w \rangle$, a (LIFO) list OPEN for open (sub-) goals is initialized with the open goals created during the construction of the inductive case analysis for $\langle \Gamma\, ;\, w \rangle$ by `ind-case-analysis` (see Section 8.3.3). In the subsequent steps, the list OPEN is iteratively processed as long as it still contains elements. In each of these steps, the tactic always deals with the first goal $\langle \Gamma'\, ;\, w' \rangle$ of OPEN. After removing $\langle \Gamma'\, ;\, w' \rangle$ from OPEN, `standard-strategy` checks whether $\langle \Gamma'\, ;\, w' \rangle$ is an order subgoal. If this is the case then the next element of OPEN is processed. Otherwise, `standard-strategy` simplifies $\langle \Gamma'\, ;\, w' \rangle$ as often as possible in consecutive applications of the tactics `cleanup`, `apply-axiom`, `apply-lemma` and `apply-ind-hypothesis` (see Section 8.3.2). The open goals generated during this simplification of $\langle \Gamma'\, ;\, w' \rangle$ are then inserted into OPEN, and the next open goal in OPEN is dealt with. In case that $\langle \Gamma'\, ;\, w' \rangle$ cannot be simplified by any of these tactics, `standard-strategy` creates a *new* proof state tree T' for the goal $\langle \Gamma'\, ;\, w' \rangle$. It does so by calling the inference machine operation corresponding to the `assign-name` command (see Section 7.3.2). Thereafter, `standard-strategy` is recursively invoked for $\langle \Gamma'\, ;\, w' \rangle$ in T', and having returned from this invocation, the tactic processes the next element of OPEN.

Once the list OPEN has become empty, each of the remaining open goals in T should be an order subgoal. Since there are still weight variables in T, `standard-strategy` calls the procedure `set-weights` for T. Unless the call of this procedure fails, `standard-strategy` applies the tactic `prove-order-subgoal` to each of the order subgoals. After this, `standard-strategy` terminates (successfully).

We still have to describe how the tactic `standard-strategy` proceeds if the original goal $\langle \Gamma\, ;\, w \rangle$ can actually be simplified with the tactic `simplify-goal` (see above). In this case, `standard-strategy` creates, for each open goal $\langle \Gamma'\, ;\, w' \rangle$ in T, a new proof state tree T' and then applies itself recursively to $\langle \Gamma'\, ;\, w' \rangle$ in T'. Thereafter, `standard-strategy` terminates (successfully).

In order to illustrate the tactic `standard-strategy` with a small example, we explain in the following how the tactic constructs a *complete* proof for the conjecture (8.9) from Section 8.3.3. When invoked for the goal

```
G-node root  "elem-app-3"
 { elem(x,append(l1,l2)) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(l1,x,l2)
```

at the root of the proof state tree `elem-app-3`, the tactic `standard-strategy` first attempts to simplify this goal with the tactic `simplify-goal`. Since this call fails (the conjecture is simplified), `standard-strategy` applies the tactic `ind-case-analysis` to it, which leads to the inference steps described in the example of Section 8.3.3. As was shown there, the only open goal in `elem-app-3` resulting from the construction of the inductive case analysis for (8.9) is the goal at position $1.2.1^3.2.1^2$, namely

```
G-node [1:2:1^3:2:1^2]
 { x = y,
   elem(x,cons(y,append(l1,l2))) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

This goal is not an order subgoal, and so `standard-strategy` tries to simplify it: first with the tactic `cleanup`, which fails; and then with the tactic `apply-axiom`, which effects an application of the inference rule Non-Inductive Rewriting with the defining rule `elem-3`. The goals generated in this rewrite step are

```
G-node [1:2:1^3:2:1^4]
 { def append(l1,l2),
   x = y,
   elem(x,cons(y,append(l1,l2))) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)

G-node [1:2:1^3:2:1^3:2]
 { ~def append(l1,l2),
   x = y,
   elem(x,append(l1,l2)) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

Since the definedness subgoal at position $1.2.1^3.2.1^4$ is proved by the call of the tactic `prove-def-subgoal`, the only open goal resulting from the invocation of `apply-axiom` for the goal at positon $1.2.1^3.2.1^2$ is the goal at position $1.2.1^3.2.1^3.2$. The subsequent calls of the tactics `cleanup`, `apply-axiom` and `apply-lemma` for the goal at position $1.2.1^3.2.1^3.2$ fail. Hence, `standard-strategy` invokes `apply-ind-hypothesis`. The call of this tactic succeeds, giving rise to an application of the inference rule Inductive

Subsumption to the goal at position $1.2.1^3.2.1^3.2$ with the goal at the root of `elem-app-3` (i.e. the conjecture) as the induction hypothesis. One subgoal is generated in this inference step, namely the order subgoal

```
G-node [1:2:1^3:2:1^3:2:1^2]
 { w_elem-app-3(l1,x,l2) < w_elem-app-3(cons(y,l1),x,l2),
   ~def append(l1,l2),
   x = y,
   elem(x,append(l1,l2)) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 w_elem-app-3(cons(y,l1),x,l2)
```

This order subgoal is the only remaining *open* goal in the proof state tree `elem-app-3`. Thus, `standard-strategy` applies the procedure `set-weights` to `elem-app-3`. Since $l_1 <_H$ cons(y, l_1), `set-weights` finds the tuple (1) and assigns the weight l_1 to the weight variable w_elem-app-3(l_1, x, l_2) (see above). As a consequence, the order subgoal is changed to

```
G-node [1:2:1^3:2:1^3:2:1^2]
 { l1 < cons(y,l1),
   ~def append(l1,l2),
   x = y,
   elem(x,append(l1,l2)) =/= true,
   elem(x,l1) = true,
   elem(x,l2) = true } ;
 cons(y,l1)
```

After this, `standard-strategy` calls the tactic `prove-order-subgoal` for the goal at position $1.2.1^3.2.1^3.2.1^2$, which leads to a proof of this order subgoal by an application of the inference rule <-Decomposition. At this point, the tactic `standard-strategy` has successfully completed the construction of the proof state tree `elem-app-3` for the conjecture (8.9), as there are no more open goals in `elem-app-3`.

We urge the reader to refer to Appendix E for a collection of example specifications and proof constructions involving arithmetic operations, lists (of natural numbers) and binary (search) trees. As the reader will see, this collection includes various inductive theorems which can be proved automatically by invocations of the tactic `standard-strategy`.

The Strategy Tactic `restricted-strategy`

Mainly because of the *recursive* nature of the tactic `standard-strategy` (see above), it is possible that a proof construction performed by this tactic may *diverge* in the sense that it keeps creating new proof state trees for "senseless" conjectures of increasing complexity. This may happen e.g. when a lemma necessary for a successful proof has not been introduced yet (by the user). In such situations, the user should apply the tactic `restricted-strategy` instead, which is a restricted and usually terminating variant of `standard-strategy` provided by the module `Proof-Strategies`. Roughly speaking, the major difference between these two tactics is that `restricted-strategy` is not recursive — it simply terminates whenever `standard-strategy` would create a new proof state tree and recursively apply itself to it.

Consider e.g. the conjecture `times-sx-y` from Section E.1.2, which is introduced there with the command

```
prove { times(s(x),y) = plus(times(x,y),y) } times-sx-y
```

and inductively valid w.r.t. the underlying specification. A call of `standard-strategy` for `times-sx-y` leads to a diverging proof construction that comprises a (potentially) infinite number of newly created proof state trees. However, if the tactic `restricted-strategy` is applied to `times-sx-y`, the proof construction terminates with a proof state tree that *almost* represents a proof for the given conjecture — only one open goal remains, namely

```
G-node [1:2:1^5:2:1:2:1:2:1^6]
 { ~def times(x,y),
   ~def plus(times(x,y),x),
   ~def plus(times(x,y),y),
   s(plus(times(x,y),plus(y,x))) = s(plus(times(x,y),plus(x,y))) } ;
 s(y)
```

An analysis of this goal shows that the sub-term $\mathsf{plus}(x, y)$ at position 2.1.2 in the fourth literal should be rewritten with the commutativity of plus to obtain a "tautological" goal. For this purpose, the lemma `plus-com` may be used, which was proved before but *not* activated in order to avoid non-terminating simplification processes (see Section E.1.2). All in all, the following three commands (or *interactions*) are sufficient for a complete proof of the example conjecture.

```
call restricted-strategy
apply lemma-rewrite 4 [2:1:2] plus-com 1 []
call simplify-goal
```

That is, after invoking the tactic `restricted-strategy`, the user applies the lemma `plus-com` using (the "`lemma`"-variant of) the *elementary* inference rule Non-Inductive Rewriting in an `apply` command (see Section 7.3.2). Then he can complete the proof construction conveniently by simply calling the "general purpose" simplification tactic `simplify-goal`.

Observe that Appendix E includes further example proof constructions that involve invocations of the tactic `restricted-strategy`.

8.4 Concluding Remarks

After the presentation of our approach to flexible forms of interactive and automated proof control in this chapter, the question remains as to how effectively QUODLIBET assists the user in what we call the *proof engineering* process required for proofs of non-trivial "real-life" inductive theorems.[39] Obviously, an answer to this question should be based on experimental data, and so we provide a collection of example specifications and proof constructions involving arithmetic operations, lists (of natural numbers), the mergesort sorting algorithm and binary (search) trees in Appendix E. Instead of discussing these examples in detail, we restrict ourselves to two summarizing remarks.

First of all, we would like to make a comment on the strength of the proposed strategy tactics. Appendix E includes proof constructions for a total of 80 inductive theorems, 26 of which are (simpler) domain lemmas for the axiomatized operations. The tactic `standard-strategy` is capable of automatically proving 60 of these inductive theorems. Moreover, in 15 of the remaining 20 proof constructions, the first interaction is a call of the tactic `restricted-strategy`, which performs at least the initial inductive case analysis and the proof of the base case(s). Considering that the QML routines currently provided by QUODLIBET were developed in a first and "rapid" attempt, we find these results rather encouraging.

Secondly, we think that the proof constructions given in Appendix E may be regarded as sufficient evidence for our claim that the proposed approach to flexible forms of proof control fulfills the fourth major requirement that we laid down for the development of QUODLIBET, namely the one related to proof control (see Section 7.1). As the example proof constructions show, we have achieved *flexibility* with regard to proof control in the following sense: Aside from the elementary inference rules, the user may benefit from inference operators on *intermediate* levels — such as the tactics `simplify-goal` or `ind-case-analysis` — and from the two strategy tactics as *high* level inference operators. Moreover, the use of QML routines is well integrated into the graphical user interface of XQUODLIBET (see Section 8.2.2).

[39]Refer to the last part of Section 7.1 for a discussion of our requirements with regard to proof control.

Chapter 9

Conclusion

The overall subject of this thesis is the inductive theorem prover QuodLibet which we have developed to attain two essential goals. Firstly, our prover is applicable to data types with *partial* operations, although data types like these lead to specifications that need not be sufficiently complete or may induce non-terminating rewrite relations. Secondly, by offering a high degree of flexibility with regard to the supported forms of proof control, the *tactic*-based approach to the problem of proof control developed for QuodLibet yields a *practically* useful compromise between interactive proof control on the one hand and automated proof control on the other hand.

With regard to the formal framework for inductive theorem proving presented in Part I we would like to emphasize the following points.

Our *specification language* supports adequate formalizations of data types with *partial* operations. This includes those data types that call for incomplete or even non-terminating specifications. Moreover, the language admits as axioms in specifications conditional equations (or rewrite rules) with positive and *negative* conditions. Last but not least, it has precisely defined admissibility conditions which do not presuppose termination of the rewrite relation induced by the specification and which can be easily verified for most practically relevant specifications.

The *semantic induction order* proposed in this thesis does not require the rewrite relation associated with an admissible specification to be terminating. As a consequence, QuodLibet admits proofs for various inductive theorems that express properties of non-terminating operations — unlike any other inductive theorem prover known to us. Furthermore, our induction order is well-suited for proofs by destructor induction, which is mainly due to its "semantic" character and the order literals of our formal framework.

The presented *calculus for inductive proofs* is characterized by the following features. First of all, the calculus is goal-directed. Furthermore, our applicative inference rules are based on a concept of explicitly applicable induction hypotheses and lemmas. Besides, the calculus appears to be reasonably comprehensible and hence *user-oriented*. The calculus is also *expressive* in that it provides inference rules for (user-guided) case analyses and inference rules for (user-guided) explicit instantiations of lemmas and induction hypotheses, respectively. All in all, the inference rules are roughly modeled on what we consider elementary proof steps.

Due to the proposed concept of a *proof state graph*, our framework for inductive theorem proving supports the *lazy* generation of induction hypotheses and *mutual* induction. Moreover, we have shown that applications of *unproved* lemmas are possible, and the discussion of choice points made it clear that *multiple* complete or incomplete proof attempts for the same conjecture may occur in proof state graphs. Finally, the notion of a proof graph and our major soundness result (Theorem 6.2.4) yield a useful criterion for recognizing inductive validity that is expressed in terms of graph theoretical properties of proof attempts. As a consequence, many clauses occurring in proof constructions can be identified as inductively valid long before the proof construction has been completed.

As to the software system QUODLIBET, which we discussed in Part II, the subsequent points should be mentioned.

We have shown that our inductive theorem prover may be employed as a full-scale proof checker: Firstly, the functionality of the system is such that it allows the user (i) to easily formalize numerous practically relevant data types and (ii) to construct virtually any inductive proof with the prover that is feasible within our formal framework. In particular, QUODLIBET supports navigation through proof state trees, and goal nodes may be expanded more than once, thus giving rise to choice points in proof state graphs which represent multiple proof attempts for the same conjecture. Secondly, QUODLIBET guarantees the soundness of every successfully executed command required in the axiomatization of a data type or in proof constructions. Thirdly, the graphical user interface of XQUODLIBET significantly adds to the user-friendliness of the prover by visualizing interactive proof constructions. In addition to the GUI, the system also provides a sometimes more efficiently usable command interpreter for experienced users.

Last but not least we have proposed a novel solution to the problem of proof control for our inductive theorem prover QUODLIBET that is characterized by a great deal of *flexibility* with regard to the forms of proof control the prover supports. This approach is based on (proof) tactics, for which we developed a special proof control language named QML. Since QUODLIBET provides (in addition to the elementary inference rules) various tactics ranging from "intermediate level" tactics for simplification tasks to "high level" tactics representing comprehensive inductive proof strategies, we achieve a useful compromise between ("mindless") completely interactive proof control on the one hand and ("hard-wired") automated proof control on the other hand. Moreover, practical experience of programming in QML has shown that it is comparatively easy to extend existing (strategy) tactics or to write new ones in order to implement newly developed proof heuristics.

Appendix A

The Proofs

Proof of Lemma 3.1.3. $\mathcal{GT}(sig)$ is initial in the class of all sig-algebras, and so there is only one sig-homomorphism from $\mathcal{GT}(sig)$ to $\mathcal{B}$, namely $\mathrm{eval}^{\mathcal{B}}$. Hence, we have $h_s(t^{\mathcal{A}}) = t^{\mathcal{B}}$ for all $t \in \mathcal{GT}(sig)_s$ and for all $s \in S$, which implies that $h_s(a) \in B^C_s$ for all $a \in A^C_s$. Thus, h^C_s is a function with $h^C_s\colon\ A^C_s \to B^C_s$. With the assumption that h is a sig-homomorphism, it is easy to show that h^C is a sig^C-homomorphism. As $\mathcal{GT}(sig^C)$ is initial in the class of all sig^C-algebras, we have $\mathrm{eval}^{\mathcal{B}^C} = h^C \circ \mathrm{eval}^{\mathcal{A}^C}$. Since $\mathrm{eval}^{\mathcal{B}^C}$ is a sig^C-epimorphism, h^C must be a sig^C-epimorphism as well. □

Proof of Lemma 3.1.5. Since σ is a constructor substitution, $(\mathrm{eval}^{\mathcal{A}}_{\varphi} \circ \sigma)(x) \in A^C_s$ for all $x \in X \cap V^{\mathrm{C}}$, so $\mathrm{eval}^{\mathcal{A}}_{\varphi} \circ \sigma$ is indeed a valuation of X in $\mathcal{A}$. A simple structural induction on t shows that $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t\sigma) = \mathrm{eval}^{\mathcal{A}}_{(\mathrm{eval}^{\mathcal{A}}_{\varphi} \circ \sigma)}(t)$ for all $t \in \mathcal{T}(sig, V)$. □

Proof of Lemma 3.1.8. Let $\mathcal{A}$ be a data model, and let $\mathcal{B} \in \mathrm{Mod}(spec)$. Define a family $h = (h_s)_{s \in S}$ of functions $h_s\colon\ A^C_s \to B^C_s$ by $h_s(a) = t^{\mathcal{B}}$ where $a = t^{\mathcal{A}}$ and $t \in \mathcal{GT}(sig^C)_s$. We show that h_s is well-defined. Let $t_1, t_2 \in \mathcal{GT}(sig^C)_s$ such that $t_1^{\mathcal{A}} = a = t_2^{\mathcal{A}}$. Since $\mathcal{A}$ is a data model, $\mathrm{Mod}(spec) \models t_1 = t_2$ and hence $t_1^{\mathcal{B}} = t_2^{\mathcal{B}}$. Thus we have $h_s(t_1^{\mathcal{A}}) = t_1^{\mathcal{B}} = t_2^{\mathcal{B}} = h_s(t_2^{\mathcal{A}})$. A simple argument proves that $h\colon\ \mathcal{A}^C \to \mathcal{B}^C$ is a sig^C-homomorphism. Now let $h'\colon\ \mathcal{A}^C \to \mathcal{B}^C$ be another sig^C-homomorphism from $\mathcal{A}^C$ to $\mathcal{B}^C$. An induction on $\mathcal{GT}(sig^C)$ shows that $h(t^{\mathcal{A}}) = h'(t^{\mathcal{A}})$ for all $t \in \mathcal{GT}(sig^C)$, which implies that $h(a) = h'(a)$ for all $a \in A^C$. Hence, there is exactly one sig^C-homomorphism from $\mathcal{A}^C$ to $\mathcal{B}^C$, and so $\mathcal{A}^C$ is initial in $\{\, \mathcal{B}^C \mid \mathcal{B} \in \mathrm{Mod}(spec) \,\}$.

Conversely, let $\mathcal{A}^C$ be initial in $\{\, \mathcal{B}^C \mid \mathcal{B} \in \mathrm{Mod}(spec) \,\}$. Let $\mathcal{B} \in \mathrm{Mod}(spec)$ and $h\colon\ \mathcal{A}^C \to \mathcal{B}^C$ be the unique sig^C-homomorphism from $\mathcal{A}^C$ to $\mathcal{B}^C$. If $t_1^{\mathcal{A}} = t_2^{\mathcal{A}}$ for any $t_1, t_2 \in \mathcal{GT}(sig^C)$ then $t_1^{\mathcal{B}} = h(t_1^{\mathcal{A}}) = h(t_2^{\mathcal{A}}) = t_2^{\mathcal{B}}$ due to the definition of h. As $t_1 = t_2$ is a ground equation, $t_1 = t_2$ is valid in $\mathcal{B}$. Thus, $\mathrm{Mod}(spec) \models t_1 = t_2$, and so $\mathcal{A}$ is a data model. □

Proof of Corollary 3.1.9. As both $\mathcal{A}^C$ and $\mathcal{B}^C$ are initial in $\{\, \mathcal{B}^C \mid \mathcal{B} \in \mathrm{Mod}(spec) \,\}$, they must be ($sig^C$-) isomorphic (see e.g. Ehrig & Mahr, 1985). □

Proof sketch of Lemma 3.2.3. (1) and (2) can be shown by induction on i, and (3) and (4) follow immediately from Definitions 3.2.1 and 3.2.2. □

In the following proofs, let $[t]$ denote the *sig*-congruence class of $t \in \mathcal{T}(sig, V^{\mathrm{G}})$ with respect to $\xleftrightarrow{*}_R$, i.e. $[t] = \{\, t' \in \mathcal{T}(sig, V^{\mathrm{G}}) \mid t' \xleftrightarrow{*}_R t \,\}$.

Proof of Proposition 3.2.5. (1) Let $\mathcal{M} = \mathcal{T}(sig, V^{\mathrm{G}})/\xleftrightarrow{*}_R$. By Lemma 3.2.3(4), $\longrightarrow_R$ is sort-invariant and monotonic. Thus, $\xleftrightarrow{*}_R$ is a *sig*-congruence on $\mathcal{T}(sig, V^{\mathrm{G}})$ so that $\mathcal{M} = (M, F^{\mathcal{M}})$ is actually a *sig*-algebra satisfying

- $M_s = \{\, [t] \mid t \in \mathcal{T}(sig, V^{\mathrm{G}})_s \,\}$ and $M_s^C = \{\, [t] \mid t \in \mathcal{GT}(sig^C)_s \,\}$ for all $s \in S$,
- $f^{\mathcal{M}}([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$ for all $f \in F$ and for all $t_i \in \mathcal{T}(sig, V^{\mathrm{G}})_{s_i}$ where $\alpha(f) = s_1 \ldots s_n s$ and $i = 1, \ldots, n$.

First we prove that $\mathcal{M}$ is a *sig*-model of *spec*. Let $l = r \leftarrow \Delta$ be a conditional rewrite rule in R and φ be a valuation of V in $\mathcal{M}$ such that $\mathcal{M}$ satisfies each literal in Δ with φ. We have to show that $\mathrm{eval}_\varphi^{\mathcal{M}}(l) = \mathrm{eval}_\varphi^{\mathcal{M}}(r)$. Let u be the valuation of V^{G} in $\mathcal{M}$ with $u(x) = [x]$ for all $x \in V^{\mathrm{G}}$. By the Axiom of Choice, there is an inductive substitution σ for φ such that $\varphi(x) = (\mathrm{eval}_u^{\mathcal{M}} \circ \sigma)(x) = [x\sigma]$ for all $x \in V$. Then, by Lemma 3.1.5, $\mathrm{eval}_\varphi^{\mathcal{M}}(t) = \mathrm{eval}_{(\mathrm{eval}_u^{\mathcal{M}} \circ \sigma)}^{\mathcal{M}}(t) = [t\sigma]$ for all $t \in \mathcal{T}(sig, V)$. We claim that $l\sigma \longrightarrow_R r\sigma$ with $l = r \leftarrow \Delta$: For $u = v$ in Δ we have $\mathrm{eval}_\varphi^{\mathcal{M}}(u) = \mathrm{eval}_\varphi^{\mathcal{M}}(v)$ and hence $[u\sigma] = [v\sigma]$. Since $\longrightarrow_R$ is confluent, it follows that $u\sigma \downarrow_R v\sigma$. For $\mathrm{def}(u)$ in Δ we have $\mathrm{eval}_\varphi^{\mathcal{M}}(u) \in M^C$. Thus, there is a $t \in \mathcal{GT}(sig^C)$ with $[u\sigma] = [t]$, and hence $u\sigma \downarrow_R t$. Because of $t \in \mathcal{GT}(sig^C)$, confluence of $\longrightarrow_R$ and Lemma 3.2.3(3), there must be a $\hat{u} \in \mathcal{GT}(sig^C)$ with $u\sigma \xrightarrow{*}_R \hat{u}$. For $u \neq v$ in Δ we have $\mathrm{eval}_\varphi^{\mathcal{M}}(u) \neq \mathrm{eval}_\varphi^{\mathcal{M}}(v)$. Condition (c) in Definition 3.2.4 implies that $\mathrm{eval}_\varphi^{\mathcal{M}}(u), \mathrm{eval}_\varphi^{\mathcal{M}}(v) \in M^C$. Thus, there are $\hat{u}, \hat{v} \in \mathcal{GT}(sig^C)$ such that $u\sigma \xrightarrow{*}_R \hat{u}$ and $u\sigma \xrightarrow{*}_R \hat{v}$. Since $[\hat{u}] = \mathrm{eval}_\varphi^{\mathcal{M}}(u) \neq \mathrm{eval}_\varphi^{\mathcal{M}}(v) = [\hat{v}]$, it follows that $\hat{u} \not\downarrow_R \hat{v}$ and hence $\hat{u} \not\downarrow_{R^C} \hat{v}$. Consequently, $l = r \leftarrow \Delta$ is applicable to $l\sigma$, and we obtain $[l\sigma] = [r\sigma]$. Hence, $\mathrm{eval}_\varphi^{\mathcal{M}}(l) = \mathrm{eval}_\varphi^{\mathcal{M}}(r)$.

That $\mathcal{M}$ is also a data model of *spec* can be seen as follows. A simple induction on i shows that $t_1 \longrightarrow_{R^C, i} t_2$ entails $\mathrm{Mod}(spec) \models t_1 = t_2$ for all $t_1, t_2 \in \mathcal{GT}(sig^C)$. Thus, $\mathrm{Mod}(spec) \models t_1 = t_2$ obviously holds if $t_1 \downarrow_{R^C} t_2$, which is implied by $t_1^{\mathcal{M}} = t_2^{\mathcal{M}}$ as $\longrightarrow_R$ is confluent. Hence, $\mathcal{M} \in \mathrm{DMod}(spec)$.

(2) For $s \in S$ and for $t \in \mathcal{GT}(sig^C)_s$ define h_s: $(\mathcal{GT}(sig^C)/\xleftrightarrow{*}_{R^C})_s \to M_s^C$ by $h_s([t]_C) = [t]$ where $[t]_C = \{\, t' \in \mathcal{GT}(sig^C)_s \mid t' \xleftrightarrow{*}_{R^C} t \,\}$. Then h_s is obviously well-defined and surjective, and by making use of the confluence of $\longrightarrow_R$, one easily proves that h_s is injective. Moreover, it is trivial to show that h is a sig^C-homomorphism. Therefore, $\mathcal{M}^C$ and $\mathcal{GT}(sig^C)/\xleftrightarrow{*}_{R^C}$ are sig^C-isomorphic. Hence, (2) follows from (1) and Corollary 3.1.9. □

Proof of Lemma 3.2.6. Let $\mathcal{A} \in \mathrm{DMod}(spec)$ and φ be a valuation of V^{G} in $\mathcal{A}$. Given that $t_1 \xleftrightarrow{*}_R t_2$ we have to show that $\mathrm{eval}_\varphi^{\mathcal{A}}(t_1) = \mathrm{eval}_\varphi^{\mathcal{A}}(t_2)$. First we claim that for all $i \in \mathbb{N}$ and $s_1, s_2 \in \mathcal{T}(sig, V^{\mathrm{G}})$ the following statements hold:

(i) If $s_1 \longrightarrow_{R^C, i} s_2$ then $\mathrm{eval}_\varphi^{\mathcal{A}}(s_1) = \mathrm{eval}_\varphi^{\mathcal{A}}(s_2)$.

(ii) If $s_1 \longrightarrow_{R, i} s_2$ then $\mathrm{eval}_\varphi^{\mathcal{A}}(s_1) = \mathrm{eval}_\varphi^{\mathcal{A}}(s_2)$.

We omit the simple proof of (i) and show (ii) by induction on i. If $i = 0$ or if $s_1 \longrightarrow_{R,i+1} s_2$ holds because of $s_1 \longrightarrow_{R^C} s_2$ then (ii) follows from (i). Otherwise we have $s_1 \longrightarrow_{R,i+1} s_2$ since there is a $p \in \mathrm{Pos}(s_1)$, an inductive substitution σ and a defining rule $l = r \leftarrow \Delta$ such that $s_1/p = l\sigma$, $s_2 = s_1[r\sigma]_p$ and each literal in $\Delta\sigma$ is "fulfilled" by $\longrightarrow_{R,i}$ (see Definition 3.2.2). We show that $\mathcal{A}$ satisfies each literal in $\Delta\sigma$ with φ which implies $\mathrm{eval}_\varphi^\mathcal{A}(l\sigma) = \mathrm{eval}_\varphi^\mathcal{A}(r\sigma)$ as $\mathcal{A} \in \mathrm{Mod}(\mathit{spec})$, and hence $\mathrm{eval}_\varphi^\mathcal{A}(s_1) = \mathrm{eval}_\varphi^\mathcal{A}(s_2)$. For $u = v$ in $\Delta\sigma$ we have $u \downarrow_{R,i} v$, and $\mathrm{eval}_\varphi^\mathcal{A}(u) = \mathrm{eval}_\varphi^\mathcal{A}(v)$ follows from the induction hypothesis. For $\mathrm{def}(u)$ in $\Delta\sigma$ there is a $\hat{u} \in \mathcal{GT}(\mathit{sig}^C)$ such that $u \xrightarrow{*}_{R,i} \hat{u}$, and the induction hypothesis implies that $\mathrm{eval}_\varphi^\mathcal{A}(u) = \mathrm{eval}_\varphi^\mathcal{A}(\hat{u}) = \hat{u}^\mathcal{A}$, and hence $\mathrm{eval}_\varphi^\mathcal{A}(u) \in A^C$. For $u \neq v$ in $\Delta\sigma$ there are $\hat{u}, \hat{v} \in \mathcal{GT}(\mathit{sig}^C)$ such that $u \xrightarrow{*}_{R,i} \hat{u}$, $v \xrightarrow{*}_{R,i} \hat{v}$ and $\hat{u} \not\downarrow_{R^C} \hat{v}$. Again, it follows from the induction hypothesis that $\mathrm{eval}_\varphi^\mathcal{A}(u) = \hat{u}^\mathcal{A}$ and $\mathrm{eval}_\varphi^\mathcal{A}(v) = \hat{v}^\mathcal{A}$. As $\hat{u} \not\downarrow_{R^C} \hat{v}$, we have $\hat{u} \not\downarrow_R \hat{v}$ by Lemma 3.2.3(3), and thus $\hat{u}^\mathcal{M} \neq \hat{v}^\mathcal{M}$ for $\mathcal{M} = \mathcal{T}(\mathit{sig}, V^\mathrm{G})/\xleftrightarrow{*}_R$. Now $\mathcal{A}$ is a data model of *spec*, so $\mathcal{A}^C$ is isomorphic to $\mathcal{M}^C$ by Corollary 3.1.9 and Proposition 3.2.5. Hence, $\hat{u}^\mathcal{A} \neq \hat{v}^\mathcal{A}$ which shows that $\mathrm{eval}_\varphi^\mathcal{A}(u) \neq \mathrm{eval}_\varphi^\mathcal{A}(v)$. This completes the proof of (ii), and by applying (ii) we can easily conclude $\mathrm{eval}_\varphi^\mathcal{A}(t_1) = \mathrm{eval}_\varphi^\mathcal{A}(t_2)$ from $t_1 \xleftrightarrow{*}_R t_2$.

Conversely, let $\mathrm{DMod}(\mathit{spec}) \models t_1 = t_2$ and $\mathcal{M} = \mathcal{T}(\mathit{sig}, V^\mathrm{G})/\xleftrightarrow{*}_R$. By Proposition 3.2.5, $\mathcal{M}$ is a data model of *spec*. Hence, $\mathrm{eval}_\varphi^\mathcal{M}(t_1) = \mathrm{eval}_\varphi^\mathcal{M}(t_2)$ for every valuation φ of V^G in $\mathcal{M}$. Let u be the valuation of V^G in $\mathcal{M}$ defined by $u(x) = [x]$ for all $x \in V^\mathrm{G}$. Then $[t_1] = \mathrm{eval}_u^\mathcal{M}(t_1) = \mathrm{eval}_u^\mathcal{M}(t_2) = [t_2]$, which entails that $t_1 \xleftrightarrow{*}_R t_2$. □

Proof of Theorem 3.2.7. Let $\mathcal{M} = \mathcal{T}(\mathit{sig}, V^\mathrm{G})/\xleftrightarrow{*}_R$ and u be the valuation of V^G in $\mathcal{M}$ with $u(x) = [x]$ for all $x \in V^\mathrm{G}$. Since $\mathcal{M} \in \mathrm{DMod}(\mathit{spec})$ by Proposition 3.2.5, we still have to show that, given a *sig*-algebra $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$ and a valuation φ of V^G in $\mathcal{A}$, there is a unique *sig*-homomorphism $h\colon \mathcal{M} \to \mathcal{A}$ such that $\varphi = h \circ u$, i.e. $\varphi(x) = h([x])$ for all $x \in V^\mathrm{G}$. By Lemma 3.2.6, $t_1 \xleftrightarrow{*}_R t_2$ implies $\mathcal{A} \models t_1 = t_2$ which means that $\mathrm{eval}_\varphi^\mathcal{A}(t_1) = \mathrm{eval}_\varphi^\mathcal{A}(t_2)$ for all $t_1, t_2 \in \mathcal{T}(\mathit{sig}, V^\mathrm{G})$. Thus we have $\xleftrightarrow{*}_R \subseteq \ker(\mathrm{eval}_\varphi^\mathcal{A})$, and by the Homomorphism Theorem (see Ehrig & Mahr, 1985) we obtain a *sig*-homomorphism $h\colon \mathcal{M} \to \mathcal{A}$ such that $\mathrm{eval}_\varphi^\mathcal{A}(t) = (h \circ \mathrm{nat})(t) = h([t])$ for all $t \in \mathcal{T}(\mathit{sig}, V^\mathrm{G})$. In particular, $\varphi = h \circ u$. Now let $h'\colon \mathcal{M} \to \mathcal{A}$ be another *sig*-homomorphism with $\varphi = h' \circ u$. A simple induction on t proves that $h'(t) = h(t)$ for all $t \in \mathcal{T}(\mathit{sig}, V^\mathrm{G})$. Hence, h is unique. □

The following lemma is applied in the proof of Theorem 3.2.8.

Lemma A.0.1 *Let spec* $= (\mathit{sig}, C, R)$ *be an admissible specification, and let* Γ *be a clause. Then* $\mathcal{M}(\mathit{spec}) \models \Gamma$ *iff for every inductive substitution* σ *there is a literal* λ *in* Γ *such that*

(1) λ *is of the form* $t_1 = t_2$ *and* $t_1\sigma \downarrow_R t_2\sigma$ *or*

(2) λ *is of the form* $\mathrm{def}(t)$ *and* $t\sigma \xrightarrow{*}_R \hat{t}$ *for some* $\hat{t} \in \mathcal{GT}(\mathit{sig}^C)$ *or*

(3) λ *is of the form* $t_1 \neq t_2$ *and* $t_1\sigma \not\downarrow_R t_2\sigma$ *or*

(4) λ *is of the form* $\neg\mathrm{def}(t)$ *and not* $t\sigma \xrightarrow{*}_R \hat{t}$ *for every* $\hat{t} \in \mathcal{GT}(\mathit{sig}^C)$

Proof sketch of Lemma A.0.1. Essentially, the lemma follows from the fact that for every valuation φ of V in $\mathcal{M}(spec)$ there is an inductive substitution σ (and vice versa) such that $\mathrm{eval}_{\varphi}^{\mathcal{M}(spec)}(t) = [t\sigma]$ for all $t \in \mathcal{T}(sig, V)$ (see the proof of Proposition 3.2.5). $\square$

Proof of Theorem 3.2.8. Let $\mathcal{A} \in \mathrm{DMod}(spec)$ and φ be a valuation of V in $\mathcal{A}$. We have to show that there is a literal λ in Γ such that $\mathcal{A}$ satisfies λ with φ. By the Axiom of Choice, there is an inductive substitution σ such that $\sigma(x) = t$ for some $t \in \mathcal{GT}(sig^C)$ with $t^{\mathcal{A}} = \varphi(x)$ for every $x \in V^{\mathrm{C}}$, and $\sigma(x) = x$ for every $x \in V^{\mathrm{G}}$. Now (*) $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t) = \mathrm{eval}_{\varphi}^{\mathcal{A}}(t\sigma)$ for all $t \in \mathcal{T}(sig, V)$, which can be shown by induction on t. Since $\mathcal{M}(spec) \models \Gamma$, there is a literal λ in Γ for σ such that one of the cases (1) to (3) of Lemma A.0.1 must hold – case (4) contradicts the assumption made for Γ. In case of (1), λ is of the form $t_1 = t_2$ and $\mathcal{A} \models t_1\sigma = t_2\sigma$ by Lemma 3.2.6. Hence, $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1\sigma) = \mathrm{eval}_{\varphi}^{\mathcal{A}}(t_2\sigma)$, and by (*) it follows that $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1) = \mathrm{eval}_{\varphi}^{\mathcal{A}}(t_2)$. In case of (2), λ is of the form $\mathrm{def}(t)$ and $\mathcal{A} \models t\sigma = \hat{t}$ for some $\hat{t} \in \mathcal{GT}(sig^C)$ by Lemma 3.2.6. Thus, $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t) = \mathrm{eval}_{\varphi}^{\mathcal{A}}(t\sigma) = \mathrm{eval}_{\varphi}^{\mathcal{A}}(\hat{t}) = \hat{t}^{\mathcal{A}}$, and $\mathcal{A}$ satisfies $\mathrm{def}(t)$ with φ. In case of (3), λ is of the form $t_1 \neq t_2$ and $t_1\sigma \not\downarrow_R t_2\sigma$. The assumption for Γ and the preceding case imply that $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1) = \hat{t_1}^{\mathcal{A}}$ and $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_2) = \hat{t_2}^{\mathcal{A}}$ for some $\hat{t_1}, \hat{t_2} \in \mathcal{GT}(sig^C)$. Obviously, we have $\hat{t_1} \not\downarrow_{R^C} \hat{t_2}$, and hence $\hat{t_1}^{\mathcal{M}(spec)} \neq \hat{t_2}^{\mathcal{M}(spec)}$. By Proposition 3.2.5(2), we obtain $\hat{t_1}^{\mathcal{A}} \neq \hat{t_2}^{\mathcal{A}}$, and thus $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1) \neq \mathrm{eval}_{\varphi}^{\mathcal{A}}(t_2)$. $\square$

Proof of Theorem 3.3.2. For a proof of the theorem we will apply Theorem 68(I) of Wirth (1995), which guarantees ω-shallow confluence of R, V^{G}. By Corollary 23 of Wirth (1995), ω-shallow confluence of R, V^{G} is sufficient for confluence of $\longrightarrow_R$.

The rewrite relation $\longrightarrow_R$ is not changed when, for a term $t \in \mathcal{T}(sig^C, V^{\mathrm{C}})$, the literal $\mathrm{def}(t)$ is added to the condition literals of a rewrite rule. Thus, w.l.o.g., each $l = r \leftarrow \Delta$ in R satisfies the following (simplified) weak normality condition: For each $t_1 = t_2$ in Δ there is an $i \in \{1, 2\}$ such that t_i is a $\longrightarrow_R$-irreducible ground term or $\mathrm{def}(t_i)$ occurs in Δ. This implies the quasi-normality of Definition 58 of Wirth (1995) which is required for the application of Theorem 68(I) – just like conservative constructors (which follows directly from our definition of constructor rule), (a weak form of) left-linearity of R and confluence of $\longrightarrow_{R^C}$.

Theorem 68(I) requires us to show some sophisticated ω-shallow joinability properties for the critical pairs of the form $(0,1)$, $(1,0)$ or $(1,1)$. Since we want to make use of the assumed complementarity of the critical pairs for showing that their conditions are infeasible, we do not really need the complete definitions of these joinability properties, but only show that the conditions under which the critical pairs must be joined are never satisfied.

Let us consider the critical pairs $((t_0\sigma, \Delta_0\sigma, a_0),\ (t_1\sigma, \Delta_1\sigma, a_1))$ of the form $(0,1)$ or $(1,0)$ at first. The conditions of the ω-shallow joinability properties allow us to assume that $(\Delta_0\Delta_1)\sigma\tau$ is "fulfilled" by $\longrightarrow_R$ (see Definition 3.2.2) for those inductive substitutions τ for which some special form of joinability of $t_0\sigma\tau$ and $t_1\sigma\tau$ is to be given. Due to the assumed complementarity, there are two possible cases:

Case 1. There are $u,v\in\mathcal{T}(sig,V)$ and an $i\in\{0,1\}$ such that $u=v$ or $v=u$ occurs in $\varDelta_i\sigma$ and $u\neq v$ occurs in $\varDelta_{1-i}\sigma$:
Under this assumption there are some $\hat{u},\hat{v}\in\mathcal{GT}(sig^C)$ such that $\hat{u}\overset{*}{\longleftarrow}_R u\tau\overset{*}{\longrightarrow}_R\circ\overset{*}{\longleftarrow}_R v\tau\overset{*}{\longrightarrow}_R\hat{v}$ and $\hat{u}\not\downarrow_{R^C}\hat{v}$. By Lemma 3.2.3(3) and by Claim 1 below we get $\hat{u}\overset{*}{\longleftarrow}_{R^C} u\tau\overset{*}{\longrightarrow}_{R^C}\circ\overset{*}{\longleftarrow}_{R^C} v\tau\overset{*}{\longrightarrow}_{R^C}\hat{v}$ and then by confluence of $\longrightarrow_{R^C}$ the contradictory $\hat{u}\downarrow_{R^C}\hat{v}$.
Claim 1. We have $u\tau,v\tau\in\mathcal{GT}(sig^C)$.
Proof of Claim 1. Since the critical pair is of the form $(0,1)$ or $(1,0)$, one of the rules generating the critical pair must be a constructor rule. Thus, since u and v occur in both instantiated condition lists, they occur also in the instantiated condition list of the constructor rule, such that we have $u,v\in\mathcal{T}(sig^C,V^{\mathrm{C}})$ by Definition 3.2.1 and because σ is a constructor substitution, and thus $u\tau,v\tau\in\mathcal{GT}(sig^C)$ because τ is an inductive substitution.

Case 2. There are $t,\hat{u},\hat{v}\in\mathcal{T}(sig,V)$ such that $\hat{u},\hat{v}$ are distinct $\longrightarrow_R$-irreducible ground terms, $t=\hat{u}$ or $\hat{u}=t$ occurs in $\varDelta_0\sigma$ and $t=\hat{v}$ or $\hat{v}=t$ occurs in $\varDelta_1\sigma$:
Under this assumption we have $\hat{u}\overset{*}{\longleftarrow}_R t\tau\overset{*}{\longrightarrow}_R\hat{v}$. By Lemma 3.2.3(3) and by Claim 2 below we get $\hat{u}\overset{*}{\longleftarrow}_{R^C} t\tau\overset{*}{\longrightarrow}_{R^C}\hat{v}$ and then by the assumed confluence of $\longrightarrow_{R^C}$ the contradictory $\hat{u}\downarrow_{R^C}\hat{v}$.
Claim 2. We have $t\tau\in\mathcal{GT}(sig^C)$.
Proof of Claim 2. Since the critical pair is of the form $(0,1)$ or $(1,0)$, one of the rules generating the critical pair must be a constructor rule. Thus, since t occurs in both instantiated condition lists, it occurs also in the instantiated condition list of the constructor rule, such that we have $t\in\mathcal{T}(sig^C,V^{\mathrm{C}})$ by Definition 3.2.1 and because σ is a constructor substitution, and thus $t\tau\in\mathcal{GT}(sig^C)$ because τ is an inductive substitution.

Let us now consider the critical pairs $((t_0\sigma,\varDelta_0\sigma,a_0),\ (t_1\sigma,\varDelta_1\sigma,a_1))$ of the form $(1,1)$. The conditions of the ω-shallow joinability properties allow us to assume the following for those $n_0,n_1\in\mathbb{N}$ and inductive substitution τ for which some special form of joinability of $t_0\sigma\tau$ and $t_1\sigma\tau$ is to be given: For $i\in\{0,1\}$, $\varDelta_i\sigma\tau$ is “fulfilled” by $\longrightarrow_{R,n_i}$, and R,V^{G} is ω-shallow confluent up to $\omega{+}n_0{+}n_1$. Due to the assumed complementarity, there are two possible cases:

Case 1. There are $u_0,u_1\in\mathcal{T}(sig,V)$ and an $i\in\{0,1\}$ such that $u_0=u_1$ or $u_1=u_0$ occurs in $\varDelta_i\sigma$ and $u_0\neq u_1$ occurs in $\varDelta_{1-i}\sigma$:
Since $u_0=u_1$ occurs in $\varDelta_i\sigma$, by the above statement there is some $j\in\{0,1\}$ such that $\mathrm{def}(u_j)$ occurs in $\varDelta_i\sigma$ or u_j is a $\longrightarrow_R$-irreducible ground term. Thus, by the above fulfilledness, there are some $u_0',u_1',t'\in\mathcal{GT}(sig^C)$ such that $u_0'\overset{*}{\longleftarrow}_{R,n_{1-i}} u_0\tau\overset{*}{\longrightarrow}_{R,n_i}\circ\overset{*}{\longleftarrow}_{R,n_i} u_1\tau\overset{*}{\longrightarrow}_{R,n_{1-i}} u_1'$, $u_0'\not\downarrow_{R^C} u_1'$, and $u_j\tau\overset{*}{\longrightarrow}_{R,n_i} t'$ or $u_j\tau$ is $\longrightarrow_R$-irreducible. By Lemma 69(4) of Wirth (1995) and the ω-shallow confluence of R,V^{G} up to $\omega{+}n_0{+}n_1$ this implies $u_0'\downarrow_{R,n_i} u_1'$, which by Lemma 3.2.3(3) implies the contradictory $u_0'\downarrow_{R^C} u_1'$.
Case 2. There are $t,\hat{u},\hat{v}\in\mathcal{T}(sig,V)$ such that $\hat{u},\hat{v}$ are distinct $\longrightarrow_R$-irreducible ground terms, $t=\hat{u}$ or $\hat{u}=t$ occurs in $\varDelta_0\sigma$ and $t=\hat{v}$ or $\hat{v}=t$ occurs in $\varDelta_1\sigma$:
In this case we have $\hat{u}\overset{*}{\longleftarrow}_{R,n_0} t\tau\overset{*}{\longrightarrow}_{R,n_1}\hat{v}$. By the ω-shallow confluence of R,V^{G} up to $\omega{+}n_0{+}n_1$ this implies the contradictory $\hat{u}\overset{*}{\longrightarrow}_{R,n_1}\circ\overset{*}{\longleftarrow}_{R,n_0}\hat{v}$. □

Proof of Theorem 3.4.2. Let $\mathcal{A} = ((A_s)_{s\in S_1}, (f^{\mathcal{A}})_{f\in F_1})$ be a data model of $spec_1$, and let φ be a valuation of V_1 in $\mathcal{A}$. We have to show that $\mathcal{A}$ satisfies Γ with φ. Let $\mathcal{B} = ((B_s)_{s\in S_0}, (f^{\mathcal{B}})_{f\in F_0})$ be the sig_0-reduct of $\mathcal{A}$, i.e. $B_s = A_s$ for each $s \in S_0$ and $f^{\mathcal{B}} = f^{\mathcal{A}}$ for each $f \in F_0$. Now $\hat{\varphi} := \varphi|_{V_0}$ is a valuation of V_0 in $\mathcal{B}$ such that $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t) = \mathrm{eval}^{\mathcal{B}}_{\hat{\varphi}}(t)$ for every $t \in \mathcal{T}(sig_0, V_0)$. Due to condition (4) in Definition 3.4.1, $\mathcal{GT}(sig_1^{C_1}) = \mathcal{GT}(sig_0^{C_0})$ for each $s \in S_0$, and so we have $A_s^{C_1} = B_s^{C_0}$ for each $s \in S_0$. Moreover, Γ is a clause over sig_0 and V_0. Thus, we obtain: (*) $\mathcal{A}$ satisfies Γ with φ iff $\mathcal{B}$ satisfies Γ with $\hat{\varphi}$. Assume that $\mathcal{B} \in \mathrm{DMod}(spec_0)$. Since $\mathrm{DMod}(spec_0) \models \Gamma$, $\mathcal{B}$ satisfies Γ with $\hat{\varphi}$. Hence, by (*), $\mathcal{A}$ satisfies Γ with φ.

We still have to show that $\mathcal{B}$ is a data model of $spec_0$. Since $\mathcal{A}$ is a (sig_1)-model of $spec_1$, it follows with an argument similar to (*) that each rewrite rule in R_0 is valid in $\mathcal{B}$. Thus, $\mathcal{B}$ is a (sig_0)-model of $spec_0$. Let $t_1, t_2 \in \mathcal{GT}(sig_0^{C_0})$ such that $t_1^{\mathcal{B}} = t_2^{\mathcal{B}}$. Then $t_1^{\mathcal{A}} = t_2^{\mathcal{A}}$, which entails $t_1 \downarrow_{R_1^{C_1}} t_2$ by Proposition 3.2.5(2). Using condition (5) in Definition 3.4.1 we obtain $t_1 \downarrow_{R_0^{C_0}} t_2$. This implies that $\mathrm{Mod}(spec_0) \models t_1 = t_2$ as is easily shown. Hence, $\mathcal{B}$ is a data model of $spec_0$. □

Proof of Lemma 4.1.1. If $\mathcal{A} \not\models \Gamma$ then there is a valuation φ' of V in $\mathcal{A}$ such that $\mathcal{A}$ does not satisfy Γ with φ'. By the Axiom of Choice, there is an inductive substitution σ such that $\sigma(x) = t$ for some $t \in \mathcal{GT}(sig^C)$ with $t^{\mathcal{A}} = \varphi'(x)$ for every $x \in V^{\mathrm{C}}$, and $\sigma(x) = x$ for every $x \in V^{\mathrm{G}}$. Let $\varphi = \varphi'|_{V^{\mathrm{G}}}$. A simple induction on t shows that, for any $t \in \mathcal{T}(sig, V)$, $\mathrm{eval}^{\mathcal{A}}_{\varphi'}(t) = \mathrm{eval}^{\mathcal{A}}_{\varphi}(t\sigma)$, which implies that $\mathcal{A}$ does not satisfy $\Gamma\sigma$ with φ.

Conversely, assume $\mathcal{A} \models \Gamma$. Let σ be an inductive substitution and φ be a valuation of V^{G} in $\mathcal{A}$. We have to show that $\mathcal{A}$ satisfies $\Gamma\sigma$ with φ. By the Axiom of Choice, there is a valuation φ' of V in $\mathcal{A}$ with $\varphi'(x) = \mathrm{eval}^{\mathcal{A}}_{\varphi}(x\sigma)$ for every $x \in V$. Again we have (*) $\mathrm{eval}^{\mathcal{A}}_{\varphi'}(t) = \mathrm{eval}^{\mathcal{A}}_{\varphi}(t\sigma)$ for all $t \in \mathcal{T}(sig, V)$. Because of $\mathcal{A} \models \Gamma$, $\mathcal{A}$ satisfies Γ with φ'. With (*) it is easily seen that $\mathcal{A}$ satisfies $\Gamma\sigma$ with φ. □

Proof of Lemma 4.2.6. Since *spec* is an admissible specification with free constructors, $R^C = \emptyset$ and, by Proposition 3.2.5(2), the data reduct $\mathcal{A}^C$ of $\mathcal{A}$ and $\mathcal{GT}(sig^C)$ are isomorphic. Therefore, (*) for all $a \in \mathcal{A}^C$ there is a unique $t \in \mathcal{GT}(sig^C)$ with $t^{\mathcal{A}} = a$.

Obviously, $\leq_{\mathcal{A}}$ is reflexive on A, and using (*) one can easily show that $\leq_{\mathcal{A}}$ is also antisymmetric and transitive. Hence, $\leq_{\mathcal{A}}$ is a partial order on A. By the definitions of $\leq_{\mathcal{A}}$ and $<_{\mathcal{A}}$ and by (*), any descending chain $a_1 >_{\mathcal{A}} a_2 >_{\mathcal{A}} \ldots$ of elements in A yields a corresponding descending chain $|t_1| > |t_2| > \ldots$ of natural numbers ($t_i^{\mathcal{A}} = a_i$). As $\leq$ is well-founded on $\mathbb{N}$, the latter chain is finite, and so must be the former one. Hence, $\leq_{\mathcal{A}}$ is well-founded on A. □

Proof of Theorem 4.2.8. The theorem is an immediate consequence of Definition 4.2.7, Lemma 4.2.6 and the fact that the length of weights is bounded by k_0 (see Section 2.4). □

Proof of Lemma 5.1.2. Obvious. □

Proof of Lemma 5.2.1. Having no subgoals these inference rules are obviously safe. Let $\mathcal{A} \in \mathrm{DMod}(spec)$. To prove that they are also sound we have to show for each of them that there is no $\mathcal{A}$-counterexample for the goal of the rule:

Complementary Literals: It is easy to show that any clause containing complementary literals is valid in any sig-algebra $\mathcal{A}$. Hence there is no $\mathcal{A}$-counterexample for the goal of the rule.

$\neq$-Tautology: Let $t_1, t_2 \in \mathcal{T}(sig^C, V^C)$ such that t_1 and t_2 are not unifiable. Assume there is an $\mathcal{A}$-counterexample of the form $(\langle \Gamma,\, t_1 \neq t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$. Then $t_1\sigma \neq t_2\sigma$ cannot be valid in $\mathcal{A}$, which implies that $t_1\sigma^{\mathcal{A}} = t_2\sigma^{\mathcal{A}}$. As (1) $t_1\sigma, t_2\sigma \in \mathcal{GT}(sig^C)$, (2) $\mathcal{A} \in \mathrm{DMod}(spec)$ and (3) $spec$ has free constructors, it follows that $t_1\sigma = t_2\sigma$. Hence, t_1 and t_2 must be unifiable which contradicts the assumption made above.

$<$-Tautology: By the definition of $<^{\mathrm{lex}}_{\mathcal{A}}$ we have $\mathrm{eval}^{\mathcal{A}}_{\varphi}() <^{\mathrm{lex}}_{\mathcal{A}} \mathrm{eval}^{\mathcal{A}}_{\varphi}(u_1, \ldots, u_k)$ for any valuation φ of V in $\mathcal{A}$ given that $k > 0$. Therefore, there is no $\mathcal{A}$-counterexample for the goal $\langle \Gamma,\, () < (u_1, \ldots, u_k)\,, \Delta\,;\, w\rangle$ of the rule. □

Proof of Lemma 5.2.3. By Lemma 5.1.2, both inference rules are safe. We now prove the soundness of the rules. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

$=$-Decomposition: If $t_1 = t_2$ then $k = 0$ and the clause $t_1 = t_2$ is inductively valid w.r.t. $spec$. Hence, there can be no $\mathcal{A}$-counterexample for the goal of the rule. Otherwise $\mathrm{MinDifPos}(t_1, t_2) \neq \emptyset$. Let $(\langle \Gamma,\, t_1 = t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ be an $\mathcal{A}$-counterexample. We claim there must be an $i_0 \in \{1, \ldots, k\}$ such that $(\langle\, u_{i_0} = v_{i_0}\,, \Gamma,\, t_1 = t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Assume the claim is false. Then (*) $\mathcal{A}$ satisfies $u_i\sigma = v_i\sigma$ with φ for $i = 1, \ldots, k$. Let $p \in \mathrm{MinDifPos}(t_1, t_2)$. If there exists an $i \in \{1, \ldots, k\}$ such that $(\, t_1/p = t_2/p\,) =_{\mathrm{lit}} (\, u_i = v_i\,)$ then $\mathrm{eval}^{\mathcal{A}}_{\varphi}((t_1/p)\sigma) = \mathrm{eval}^{\mathcal{A}}_{\varphi}((t_2/p)\sigma)$ due to (*). If not, $t_1/p \neq t_2/p$ occurs in Γ, Δ. Since $\mathcal{A}$ does not satisfy $\Gamma\sigma, \Delta\sigma$ with φ, we also have $\mathrm{eval}^{\mathcal{A}}_{\varphi}((t_1/p)\sigma) = \mathrm{eval}^{\mathcal{A}}_{\varphi}((t_2/p)\sigma)$. All in all, $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t_1\sigma) = \mathrm{eval}^{\mathcal{A}}_{\varphi}(t_2\sigma)$, and so $\mathcal{A}$ satisfies $t_1\sigma = t_2\sigma$ with φ. Hence, $(\langle \Gamma,\, t_1 = t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is not an $\mathcal{A}$-counterexample, which yields a contradiction (see above). Therefore, there must be an $\mathcal{A}$-counterexample $(\langle\, u_{i_0} = v_{i_0}\,, \Gamma,\, t_1 = t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$, and clearly $(\langle\, u_{i_0} = v_{i_0}\,, \Gamma,\, t_1 = t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi) \approx_{\mathcal{A}} (\langle \Gamma,\, t_1 = t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$.

def-Decomposition: If $t \in \mathcal{T}(sig^C, V^C)$ then $k = 0$ and the clause $\mathrm{def}(t)$ is inductively valid w.r.t. $spec$. Hence, there can be no $\mathcal{A}$-counterexample for the goal of the rule. Otherwise $\mathrm{MinNonCPos}(t) \neq \emptyset$. Let $(\langle \Gamma, \mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ be an $\mathcal{A}$-counterexample. We claim there must be an $i_0 \in \{1, \ldots, k\}$ such that $(\langle\, \mathrm{def}(u_{i_0})\,, \Gamma,\, \mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Assume the claim is false. Then (*) $\mathcal{A}$ satisfies $\mathrm{def}(u_i\sigma)$ with φ for $i = 1, \ldots, k$. Let $p \in \mathrm{MinNonCPos}(t)$. If there exists an $i \in \{1, \ldots, k\}$ such that $t/p = u_i$ then $\mathrm{eval}^{\mathcal{A}}_{\varphi}((t/p)\sigma) \in A^C$ due to (*). If not, $\neg\mathrm{def}(t/p)$ occurs in Γ, Δ. Since $\mathcal{A}$ does not satisfy $\Gamma\sigma, \Delta\sigma$ with φ, we also have $\mathrm{eval}^{\mathcal{A}}_{\varphi}((t/p)\sigma) \in A^C$. All in all, $\mathrm{eval}^{\mathcal{A}}_{\varphi}(t\sigma) \in A^C$. Hence, $\mathcal{A}$ satisfies $\mathrm{def}(t\sigma)$ with φ, and $(\langle \Gamma,\, \mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is not an $\mathcal{A}$-counterexample. This, however, yields a contradiction (see above). Therefore, there must be an $\mathcal{A}$-counterexample $(\langle\, \mathrm{def}(u_{i_0})\,, \Gamma,\, \mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$, and obviously we also have $(\langle\, \mathrm{def}(u_{i_0})\,, \Gamma,\, \mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi) \approx_{\mathcal{A}} (\langle \Gamma,\, \mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$. □

Proof of Lemma 5.2.5. That the inference rule is safe follows from Lemma 5.1.2. We still have to prove the soundness of the rule. Assume that $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$.

If $t_1, t_2 \in \mathcal{T}(\mathit{sig}^C, V^{\mathrm{C}})$ then $k = 0$ and $\hat{t}_i = t_i$ for $i \in \{1,2\}$. Due to $|\hat{t}_1| < |\hat{t}_2|$ and $|\hat{t}_1|_x \le |\hat{t}_2|_x$ for every $x \in V^{\mathrm{C}}$, the clause $t_1 < t_2$ is inductively valid w.r.t. *spec* then. Thus, there can be no $\mathcal{A}$-counterexample for the goal of the rule in this case.

Otherwise, $\bigcup_{i=1}^{2} \mathrm{MinNonCPos}(t_i) \neq \emptyset$. Suppose that $(\langle \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. We are going to show that there is an $i_0 \in \{1, \ldots, k\}$ such that $(\langle\, \mathrm{def}(u_{i_0})\,, \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Assume the claim is false. Since $\mathcal{A}$ does not satisfy $(\Gamma,\, t_1 < t_2\,, \Delta)\sigma$ with φ, it follows that (*) $\mathcal{A}$ satisfies $\mathrm{def}(u_i\sigma)$ with φ for $i = 1, \ldots, k$. Let $i \in \{1,2\}$ and $p \in \mathrm{MinNonCPos}(t_i)$. If there is a $j \in \{1, \ldots, k\}$ such that $t_i/p = u_j$ then $\mathrm{eval}_{\varphi}^{\mathcal{A}}((t_i/p)\sigma) \in A^C$ by (*). If not, $\neg\mathrm{def}(t_i/p)$ occurs in Γ, Δ. Since $\mathcal{A}$ does not satisfy $\Gamma\sigma, \Delta\sigma$ with φ, we have $\mathrm{eval}_{\varphi}^{\mathcal{A}}((t_i/p)\sigma) \in A^C$ again. All in all, (**) $\mathrm{eval}_{\varphi}^{\mathcal{A}}((t_i/p)\sigma) \in A^C$ for $i \in \{1,2\}$ and $p \in \mathrm{MinNonCPos}(t_i)$.

Let us now define an inductive substitution τ using the following case analysis: (1) Let $x \in \{\, \hat{t}_i/p \mid i \in \{1,2\} \wedge p \in \mathrm{MinNonCPos}(t_i)\,\}$. Then $x \in V^{\mathrm{C}}$ and there is an $i \in \{1,2\}$ and a $p \in \mathrm{MinNonCPos}(t_i)$ such that $\hat{t}_i/p = x$. By (**), $\mathrm{eval}_{\varphi}^{\mathcal{A}}((t_i/p)\sigma) \in A^C$ and so there is a $\bar{t} \in \mathcal{GT}(\mathit{sig}^C)$ such that $\mathrm{eval}_{\varphi}^{\mathcal{A}}((t_i/p)\sigma) = \bar{t}^{\mathcal{A}}$. We define $\tau(x) = \bar{t}$. (2) If $x \in V \backslash \{\, \hat{t}_i/p \mid i \in \{1,2\} \wedge p \in \mathrm{MinNonCPos}(t_i)\,\}$ then $\tau(x) = x\sigma$.

Note that τ is well-defined: If there are $i, j \in \{1,2\}$, $p \in \mathrm{MinNonCPos}(t_i)$ and $q \in \mathrm{MinNonCPos}(t_j)$ such that $\hat{t}_i/p = x = \hat{t}_j/q$ then $t_i/p = t_j/q$. This is guaranteed by the definition of C-fronts and the applicability conditions of the inference rule. Moreover, since *spec* has free constructors there is exactly one $\bar{t} \in \mathcal{GT}(\mathit{sig}^C)$ such that $\mathrm{eval}_{\varphi}^{\mathcal{A}}((t_i/p)\sigma) = \bar{t}^{\mathcal{A}}$.

By structural induction one can show that (***) $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_i\sigma) = \hat{t}_i\tau^{\mathcal{A}}$ for $i \in \{1,2\}$. Since $|\hat{t}_1| < |\hat{t}_2|$ and $|\hat{t}_1|_x \le |\hat{t}_2|_x$ for every $x \in V^{\mathrm{C}}$, we have $|\hat{t}_1\tau| < |\hat{t}_2\tau|$. Using (***) and the definition of $<_{\mathcal{A}}$ we obtain that $\mathrm{eval}_{\varphi}^{\mathcal{A}}(t_1\sigma) <_{\mathcal{A}} \mathrm{eval}_{\varphi}^{\mathcal{A}}(t_2\sigma)$. Hence, $\mathcal{A}$ satisfies $(\,t_1 < t_2\,)\sigma$ with φ. We conclude that $(\langle \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is *not* an $\mathcal{A}$-counterexample. This yields a contradiction (see above). As a consequence, there must be an $\mathcal{A}$-counterexample $(\langle\, \mathrm{def}(u_{i_0})\,, \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$, and clearly we have $(\langle\, \mathrm{def}(u_{i_0})\,, \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi) \approx_{\mathcal{A}} (\langle \Gamma,\, t_1 < t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$. □

Proof of Lemma 5.2.6. Any literal removing inference rule of the form

$$\frac{\langle \Gamma, \lambda, \Delta\,;\, w\rangle}{\langle \Gamma, \Delta\,;\, w\rangle}$$

must be sound which is shown as follows. Suppose $(\langle \Gamma, \lambda, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample for the goal of the rule. Then $\mathcal{A}$ does not satisfy any of the literals in $\Gamma\sigma$ nor in $\Delta\sigma$ with φ. Hence, $(\langle \Gamma, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample for the subgoal of the rule. Moreover, $(\langle \Gamma, \Delta\,;\, w\rangle, \sigma, \varphi) \approx_{\mathcal{A}} (\langle \Gamma, \lambda, \Delta\,;\, w\rangle, \sigma, \varphi)$.

Clearly, the inference rules Multiple Literals and $<$-Removal are safe. For the safeness proofs of the other literal removing rules assume that $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$.

$=$-Removal: Let φ be any valuation of V in $\mathcal{A}$. We claim that $\mathcal{A}$ does not satisfy $t_1 = t_2$ with φ. Hence, inductive validity of $\Gamma,\, t_1 = t_2\,, \Delta$ w.r.t. *spec* implies that of Γ, Δ. Assume to the contrary that $\mathcal{A}$ satisfies $t_1 = t_2$ with φ. As $t_1, t_2 \in \mathcal{T}(\mathit{sig}^C, V^{\mathrm{C}})$ there is an inductive substitution σ such that $\mathrm{eval}_\varphi^{\mathcal{A}}(t_i) = t_i\sigma^{\mathcal{A}}$ for $i = 1, 2$. Thus, $t_1\sigma^{\mathcal{A}} = t_2\sigma^{\mathcal{A}}$. Since $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$ and *spec* has free constructors it follows that $t_1\sigma = t_2\sigma$. Hence, t_1 and t_2 are unifiable, which contradicts the second applicability condition.

$\neq$-Removal: Let φ be a valuation of V in $\mathcal{A}$. We have to show that $\mathcal{A}$ satisfies Γ, Δ with φ. We can assume that $\Gamma,\, t_1 \neq t_2\,, \Delta$ is inductively valid w.r.t. *spec*. Thus, $\mathcal{A}$ satisfies $\Gamma,\, t_1 \neq t_2\,, \Delta$ with φ. If Γ, Δ contains a literal λ such that $\mathcal{A}$ satisfies λ with φ then we are done. If $\mathcal{A}$ satisfies $t_1 \neq t_2$ with φ then there must be a $p \in \mathrm{MinDifPos}(t_1, t_2)$ such that $\mathrm{eval}_\varphi^{\mathcal{A}}(t_1/p) \neq \mathrm{eval}_\varphi^{\mathcal{A}}(t_2/p)$. Hence, $\mathcal{A}$ satisfies $t_1/p \neq t_2/p$ with φ. Because of the second applicability condition $t_1/p \neq t_2/p$ occurs in Γ, Δ. Consequently, $\mathcal{A}$ satisfies Γ, Δ with φ.

$\neg$def-Removal: Let φ be a valuation of V in $\mathcal{A}$. To show that $\mathcal{A}$ satisfies Γ, Δ with φ we assume that $\Gamma,\, \neg\mathrm{def}(t)\,, \Delta$ is inductively valid w.r.t. *spec*. Thus, $\mathcal{A}$ satisfies $\Gamma,\, \neg\mathrm{def}(t_1)\,, \Delta$ with φ. If Γ, Δ contains a literal λ such that $\mathcal{A}$ satisfies λ with φ then we are done. If $\mathcal{A}$ satisfies $\neg\mathrm{def}(t)$ with φ then $t \notin \mathcal{T}(\mathit{sig}^C, V^{\mathrm{C}})$ and there must be a $p \in \mathrm{MinNonCPos}(t)$ such that $\mathrm{eval}_\varphi^{\mathcal{A}}(t/p) \notin A^C$. Hence, $\mathcal{A}$ satisfies $\neg\mathrm{def}(t/p)$ with φ. Due to the second applicability condition $\neg\mathrm{def}(t/p)$ occurs in Γ, Δ. Therefore, $\mathcal{A}$ satisfies Γ, Δ with φ. $\square$

Proof of Lemma 5.2.7. Let $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$.

To prove the soundness of the rule we assume that $(\langle \Gamma, \lambda[t_1]_p, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. As $t_1 \dot{\neq} t_2$ occurs in Γ, Δ it follows that (*) $\mathrm{eval}_\varphi^{\mathcal{A}}(t_1\sigma) = \mathrm{eval}_\varphi^{\mathcal{A}}(t_2\sigma)$. Now $\lambda/p = t_1$ and $\mathcal{A}$ does not satisfy $(\lambda[t_1]_p)\sigma$ with φ. Because of (*), $\mathcal{A}$ does not satisfy $(\lambda[t_2]_p)\sigma$ with φ either. Therefore, $(\langle \Gamma, \lambda[t_2]_p, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample, and we have $(\langle \Gamma, \lambda[t_2]_p, \Delta\,;\, w\rangle, \sigma, \varphi) \approx_{\mathcal{A}} (\langle \Gamma, \lambda[t_1]_p, \Delta\,;\, w\rangle, \sigma, \varphi)$.

We now prove the safeness of the rule. Let φ be a valuation of V in $\mathcal{A}$. To show that $\mathcal{A}$ satisfies $\Gamma, \lambda[t_2]_p, \Delta$ with φ we can assume that $\Gamma, \lambda[t_1]_p, \Delta$ is inductively valid w.r.t. *spec*. Thus, $\mathcal{A}$ satisfies $\Gamma, \lambda[t_1]_p, \Delta$ with φ. If Γ, Δ contains a literal satisfied by $\mathcal{A}$ with φ then we are done. Otherwise, $\mathcal{A}$ does not satisfy $t_1 \dot{\neq} t_2$ with φ. Consequently, (**) $\mathrm{eval}_\varphi^{\mathcal{A}}(t_1) = \mathrm{eval}_\varphi^{\mathcal{A}}(t_2)$. Since $\mathcal{A}$ must satisfy $\lambda[t_1]_p$ with φ and $\lambda/p = t_1$, we obtain, by (**), that $\mathcal{A}$ also satisfies $\lambda[t_2]_p$ with φ and thus $\Gamma, \lambda[t_2]_p, \Delta$. $\square$

Proof of Lemma 5.2.8. Let $\mathcal{A} \in \mathrm{DMod}(\mathit{spec})$.

For the soundness proof of the rule let us assume that $(\langle \Gamma,\, t_1 \neq t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. This means in particular that $\mathcal{A}$ does not satisfy $t_1\sigma \neq t_2\sigma$ with φ. Since $t_1, t_2 \in \mathcal{T}(\mathit{sig}^C, V^{\mathrm{C}})$ and σ is an inductive substitution, $t_1\sigma^{\mathcal{A}} = t_2\sigma^{\mathcal{A}}$. By the freeness of the constructors in *spec* we obtain $t_1\sigma = t_2\sigma$. Hence, σ is a unifier of t_1 and t_2, and there is a constructor substitution μ such that (*) $\sigma|_{\mathrm{Var}(\Gamma, t_1, t_2, \Delta, w)} =$

$(\tau\mu)|_{\mathrm{Var}(\Gamma,t_1,t_2,\Delta,w)}$. Let λ be a literal in Γ, Δ. Since $\mathcal{A}$ does not satisfy $\lambda\sigma$ with φ, it follows with (*) that $\mathcal{A}$ does not satisfy $(\lambda\tau)\mu$ with φ either. Thus, $(\langle \Gamma\tau, \Delta\tau\,;\, w\tau\rangle, \mu, \varphi)$ is an $\mathcal{A}$-counterexample. We also have $(\langle \Gamma\tau, \Delta\tau\,;\, w\tau\rangle, \mu, \varphi) \approx_{\mathcal{A}} (\langle \Gamma,\, t_1 \neq t_2\,, \Delta\,;\, w\rangle, \sigma, \varphi)$, as $w\sigma = (w\tau)\mu$.

To show the safeness of the rule we can assume that $\Gamma,\, t_1 \neq t_2\,, \Delta$ is inductively valid w.r.t. *spec*. Since τ is a constructor substitution, the instance $(\Gamma,\, t_1 \neq t_2\,, \Delta)\tau$ is also inductively valid w.r.t. *spec* (see Lemma 5.3.2). As τ is a unifier of t_1 and t_2, we have $t_1\tau = t_2\tau$. Thus, the literal $t_1\tau \neq t_2\tau$ can never be satisfied. Therefore, $\Gamma\tau, \Delta\tau$ must be inductively valid w.r.t. *spec*. □

Proof of Lemma 5.2.9. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

To show that the rule is sound we assume that $(\langle \Gamma,\, \neg\mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. This implies that $\mathrm{eval}_\varphi^{\mathcal{A}}(t\sigma) \in A^C$, i.e. there is a $\hat{t} \in \mathcal{GT}(sig^C)$ such that $\mathrm{eval}_\varphi^{\mathcal{A}}(t\sigma) = \hat{t}^{\mathcal{A}}$. We define a further substitution σ' by $\sigma'(x) = \hat{t}$ and $\sigma'(y) = y\sigma$ for every $y \in V\backslash\{x\}$. Since $\hat{t} \in \mathcal{GT}(sig^C)$ and σ is an inductive substitution, σ' is also inductive. Now $x \notin \mathrm{Var}(\Gamma, \Delta, t, w)$, and so we have (*) $t\sigma' = t\sigma$, $\Gamma\sigma' = \Gamma\sigma$, $\Delta\sigma' = \Delta\sigma$ and $w\sigma' = w\sigma$. From (*) and the above assumption we can infer that $\mathcal{A}$ does not satisfy $\Gamma\sigma', \Delta\sigma'$ with φ. Moreover, $\mathrm{eval}_\varphi^{\mathcal{A}}(x\sigma') = \hat{t}^{\mathcal{A}} = \mathrm{eval}_\varphi^{\mathcal{A}}(t\sigma) = \mathrm{eval}_\varphi^{\mathcal{A}}(t\sigma')$. Hence, $\mathcal{A}$ does not satisfy $x\sigma' \neq t\sigma'$ with φ either, and we obtain that $(\langle \Gamma,\, x \neq t\,, \Delta\,;\, w\rangle, \sigma', \varphi)$ is an $\mathcal{A}$-counterexample. Besides, $(\langle \Gamma,\, x \neq t\,, \Delta\,;\, w\rangle, \sigma', \varphi) \approx_{\mathcal{A}} (\langle \Gamma,\, \neg\mathrm{def}(t)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ as $w\sigma' = w\sigma$ by (*).

We now prove the safeness of the rule. Let φ be a valuation of V in $\mathcal{A}$. To show that $\mathcal{A}$ satisfies $\Gamma,\, x \neq t\,, \Delta$ with φ we can assume that $\Gamma,\, \neg\mathrm{def}(t)\,, \Delta$ is inductively valid w.r.t. *spec*. Thus, $\mathcal{A}$ satisfies $\Gamma,\, \neg\mathrm{def}(t)\,, \Delta$ with φ. If Γ, Δ contains a literal satisfied by $\mathcal{A}$ with φ then the claim is proved. Otherwise, $\mathrm{eval}_\varphi^{\mathcal{A}}(t) \notin A^C$. As $\mathrm{eval}_\varphi^{\mathcal{A}}(x) \in A^C$ (x is a constructor variable) we obtain $\mathrm{eval}_\varphi^{\mathcal{A}}(x) \neq \mathrm{eval}_\varphi^{\mathcal{A}}(t)$. Hence, $\mathcal{A}$ satisfies $x \neq t$ with φ and thus $\Gamma,\, x \neq t\,, \Delta$. □

Proof of Lemma 5.2.10. Let $\mathcal{A} \in \mathrm{DMod}(spec)$. We first prove the soundness of both inference rules.

Tuple $<$-Reduction: Let $(\langle \Gamma,\, (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ be an $\mathcal{A}$-counterexample. We claim $(\langle\, t_1 < t_2\,, \Gamma,\, (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is also an $\mathcal{A}$-counterexample. Assume the claim is false. Since $\mathcal{A}$ does not satisfy $(\Gamma,\, (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,, \Delta)\sigma$ with φ, $\mathcal{A}$ must satisfy $(\,t_1 < t_2\,)\sigma$ with φ, which means that $\mathrm{eval}_\varphi^{\mathcal{A}}(t_1\sigma) <_{\mathcal{A}} \mathrm{eval}_\varphi^{\mathcal{A}}(t_2\sigma)$. According to the definition of $<_{\mathcal{A}}^{\mathrm{lex}}$ (see Section 2.4) this is sufficient for $\mathrm{eval}_\varphi^{\mathcal{A}}((t_1, u_1, \ldots, u_m)\sigma) <_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_\varphi^{\mathcal{A}}((t_2, v_1, \ldots, v_n)\sigma)$. Hence, $\mathcal{A}$ does satisfy the literal $(t_1, u_1, \ldots, u_m)\sigma < (t_2, v_1, \ldots, v_n)\sigma$ with φ, which contradicts the assumption that $(\langle \Gamma,\, (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Therefore, $(\langle\, t_1 < t_2\,, \Gamma,\, (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is also an $\mathcal{A}$-counterexample, which is clearly equivalent w.r.t. $\precsim_{\mathcal{A}}$.

Tuple $=$-Reduction: Let $(\langle \Gamma,\, (t, u_1, \ldots, u_m) < (t, v_1, \ldots, v_n)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ be an $\mathcal{A}$-counterexample. We claim that $(\langle \Gamma,\, (u_1, \ldots, u_m) < (v_1, \ldots, v_n)\,, \Delta\,;\, w\rangle, \sigma, \varphi)$ is also an $\mathcal{A}$-counterexample. Assume that the claim is false. Then $\mathcal{A}$ must satisfy the literal

$(u_1,\ldots,u_m)\sigma < (v_1,\ldots,v_n)\sigma$ with φ, as $\mathcal{A}$ does not satisfy $\Gamma\sigma, \Delta\sigma$ with φ. Again the definition of $<^{\text{lex}}_{\mathcal{A}}$ implies that $\mathcal{A}$ satisfies $(t,u_1,\ldots,u_m)\sigma < (t,v_1,\ldots,v_n)\sigma$ with φ then, which yields a contradiction. Hence, $(\langle \Gamma,\ (u_1,\ldots,u_m) < (v_1,\ldots,v_n)\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is also an $\mathcal{A}$-counterexample and equivalent w.r.t. $\precsim_{\mathcal{A}}$.

By Lemma 5.1.2, Tuple <-Reduction is a safe inference rule. To show that the rule Tuple =-Reduction is safe, we can assume that (*) $\Gamma,\ (t,u_1,\ldots,u_m) < (t,v_1,\ldots,v_n)\,, \Delta$ is inductively valid w.r.t. *spec*. Let φ be a valutation of V in $\mathcal{A}$. We have to show that $\mathcal{A}$ satisfies $\Gamma,\ (u_1,\ldots,u_m) < (v_1,\ldots,v_n)\,, \Delta$ with φ. Due to (*), $\mathcal{A}$ satisfies the clause $\Gamma,\ (t,u_1,\ldots,u_m) < (t,v_1,\ldots,v_n)\,, \Delta$ with φ. If Γ, Δ contains a literal satisfied by $\mathcal{A}$ with φ then we are done. Otherwise, $\mathcal{A}$ satisfies $(t,u_1,\ldots,u_m) < (t,v_1,\ldots,v_n)$ with φ. Thus, $\text{eval}^{\mathcal{A}}_{\varphi}(t,u_1,\ldots,u_m) <^{\text{lex}}_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}(t,v_1,\ldots,v_n)$. Now this implies that $\text{eval}^{\mathcal{A}}_{\varphi}(u_1,\ldots,u_m) <^{\text{lex}}_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}(v_1,\ldots,v_n)$ (see the definition of $<^{\text{lex}}_{\mathcal{A}}$). Hence, $\mathcal{A}$ satisfies $(u_1,\ldots,u_m) < (v_1,\ldots,v_n)$ with φ and therefore $\Gamma,\ (u_1,\ldots,u_m) < (v_1,\ldots,v_n)\,, \Delta$. □

Proof of Lemma 5.2.12. Safeness of the inference rule follows from Lemma 5.1.2. We still have to prove that the rule is sound. Let $\mathcal{A} \in \text{DMod}(\mathit{spec})$.

Suppose that $(\langle \Gamma,\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. We are going to show that $(\langle\, u_1 < u_2\,, \Gamma,\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample as well. Assume the claim is false. Since $\mathcal{A}$ does not satisfy $(\Gamma,\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta)\sigma$ with φ, $\mathcal{A}$ must satisfy $(u_1 < u_2)\sigma$ with φ. This means that $\text{eval}^{\mathcal{A}}_{\varphi}(u_1\sigma) <_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}(u_2\sigma)$. In particular, $\text{eval}^{\mathcal{A}}_{\varphi}(u_i\sigma) \in A^C$ for $i \in \{1,2\}$. Hence, there are $\bar{t}_1, \bar{t}_2 \in \mathcal{GT}(\mathit{sig}^C)$ such that $\text{eval}^{\mathcal{A}}_{\varphi}(u_i\sigma) = \bar{t}_i^{\mathcal{A}}$ for $i \in \{1,2\}$, and (*) $|\bar{t}_1| < |\bar{t}_2|$ by the definition of $<_{\mathcal{A}}$. We define an inductive substitution τ by $\tau(x_i) = \bar{t}_i$ for $i \in \{1,2\}$ and $\tau(x) = x\sigma$ for $x \in V \backslash \{x_1, x_2\}$. By induction on the length of the position p_i one can easily show that (**) $\text{eval}^{\mathcal{A}}_{\varphi}((t_i[u_i]_{p_i})\sigma) = \hat{t}_i\tau^{\mathcal{A}}$ for $i \in \{1,2\}$. Now (1) $(\hat{t}_i\tau)/p_i = \bar{t}_i$ for $i \in \{1,2\}$ (2) $|\hat{t}_1| \leq |\hat{t}_2|$ and (3) $|\hat{t}_1|_x \leq |\hat{t}_2|_x$ for every $x \in V^{\text{C}} \backslash \{x_1, x_2\}$. Therefore, (*) implies that $|\hat{t}_1\tau| \leq |\hat{t}_2\tau|$. Applying (**) we obtain $\text{eval}^{\mathcal{A}}_{\varphi}((t_1[u_1]_{p_1})\sigma) <_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}((t_2[u_2]_{p_2})\sigma)$. Hence, $\mathcal{A}$ satisfies $(t_1[u_1]_{p_1} < t_2[u_2]_{p_2})\sigma$ with φ so that $(\langle \Gamma,\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is *not* an $\mathcal{A}$-counterexample. This yields a contradiction (see above). As a consequence, $(\langle\, u_1 < u_2\,, \Gamma,\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Besides, we have $(\langle\, u_1 < u_2\,, \Gamma,\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta\,;\ w\rangle, \sigma, \varphi) \approx_{\mathcal{A}} (\langle \Gamma, t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \Delta\,;\ w\rangle, \sigma, \varphi)$. □

Proof of Lemma 5.2.13. By Lemma 5.1.2, the inference rule is safe. We now prove the soundness of the rule. Let $\mathcal{A} \in \text{DMod}(\mathit{spec})$.

Let $(\langle \Gamma,\ w_1 < w_3\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ be an $\mathcal{A}$-counterexample. We claim that at least one of $(\,w_1 < w_2\,, \langle \Gamma,\ w_1 < w_3\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ or $(\,w_2 < w_3\,, \langle \Gamma,\ w_1 < w_3\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample as well. Assume the claim is false. Since $\mathcal{A}$ does not satisfy the instance $(\Gamma,\ w_1 < w_3\,, \Delta)\sigma$ with φ, $\mathcal{A}$ must satisfy both $(\,w_1 < w_2\,)\sigma$ and $(\,w_2 < w_3\,)\sigma$ with φ. Hence, $\text{eval}^{\mathcal{A}}_{\varphi}(w_1\sigma) <^{\text{lex}}_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}(w_2\sigma) <^{\text{lex}}_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}(w_3\sigma)$. As $<^{\text{lex}}_{\mathcal{A}}$ is transitive, we obtain $\text{eval}^{\mathcal{A}}_{\varphi}(w_1\sigma) <^{\text{lex}}_{\mathcal{A}} \text{eval}^{\mathcal{A}}_{\varphi}(w_3\sigma)$. Thus, $\mathcal{A}$ satisfies $(\,w_1 < w_3\,)\sigma$ with φ which contradicts the assumption that $(\langle \Gamma,\ w_1 < w_3\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Therefore, the above claim is holds. It is obvious that the $\mathcal{A}$-counterexample for one of the subgoals is equivalent to $(\langle \Gamma,\ w_1 < w_3\,, \Delta\,;\ w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. □

Proof of Lemma 5.2.15. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

Let us first prove the soundness of the rule. Assume that $(\langle \Gamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. According to Definition 5.2.14 there is a $j \in \{1, \ldots, n\}$ and an inductive substitution τ such that $x\sigma = x\sigma_j\tau$ for each $x \in \mathrm{Var}(\Gamma, w)$. By structural induction on t one can easily show that (*) $t\sigma = t\sigma_j\tau$ for every $t \in \mathcal{T}(sig, V)$ with $\mathrm{Var}(t) \subseteq \mathrm{Var}(\Gamma, w)$. Let λ be a literal in Γ. Due to (*), $\lambda\sigma = \lambda\sigma_j\tau$. Hence, $\mathcal{A}$ satisfies $\lambda\sigma$ iff $\mathcal{A}$ satisfies $\lambda\sigma_j\tau$. Thus, $(\langle \Gamma\sigma_j\,;\, w\sigma_j\rangle, \tau, \varphi)$ is also an $\mathcal{A}$-counterexample. Since $w\sigma = w\sigma_j\tau$ we have $(\langle \Gamma\sigma_j\,;\, w\sigma_j\rangle, \tau, \varphi) \approx_{\mathcal{A}} (\langle \Gamma\,;\, w\rangle, \sigma, \varphi)$.

For the safeness proof of the rule assume that Γ is an inductive theorem w.r.t. *spec* and that $j \in \{1, \ldots, n\}$. Since σ_j is a constructor substitution, the instance $\Gamma\sigma_j$ is also an inductive theorem w.r.t. *spec* by Lemma 5.3.2. $\square$

Proof of Lemma 5.2.17. By Lemma 5.1.2, the inference rule is safe. We now prove the soundness of the rule. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

Suppose $(\langle \Gamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. If $\mathcal{A}$ does not satisfy $\lambda_i\sigma$ for $i = 1, \ldots, n$ then $\mathcal{A}$ does not satisfy $\Lambda\sigma$ with φ so that $(\langle \Lambda, \Gamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Otherwise, there must be an $i \in \{1, \ldots, n\}$ such that $\mathcal{A}$ satisfies $\lambda_i\sigma$ with φ. Assume w.l.o.g. that i is the least number with this property, i.e. $\mathcal{A}$ does not satisfy $\lambda_j\sigma$ with φ for $j = 1, \ldots, i-1$. Hence, $\mathcal{A}$ does not satisfy $\Lambda_i\sigma$ with φ so that $(\langle \Lambda_i, \Gamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. In both cases, we have an $\mathcal{A}$-counterexample for a subgoal of the rule which is clearly equivalent to $(\langle \Lambda, \Gamma\,;\, w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. $\square$

Proof of Lemma 5.3.2. Let φ be a valuation of V in $\mathcal{A}$. Suppose $\mathrm{DefCond}(\mu, \Gamma) = \neg\mathrm{def}(x_1\mu), \ldots, \neg\mathrm{def}(x_n\mu)$ for some $n \in \mathbb{N}$ and $\{x_1, \ldots, x_n\} \subseteq V^{\mathrm{C}} \cap \mathrm{Var}(\Gamma)$. We have to show that $\mathcal{A}$ satisfies $\Gamma\mu, \neg\mathrm{def}(x_1\mu), \ldots, \neg\mathrm{def}(x_n\mu)$ with φ.

If there is an $i \in \{1, \ldots, n\}$ such that $\mathcal{A}$ satisfies $\neg\mathrm{def}(x_i\mu)$ with φ then the claim is proved. Otherwise, $\mathcal{A}$ satisfies $\mathrm{def}(x_i\mu)$ with φ for $i = 1, \ldots, n$. This means that (*) $\mathrm{eval}_\varphi^{\mathcal{A}}(x_i\mu) \in A^C$ for $i = 1, \ldots, n$. We define another valuation ψ of V in $\mathcal{A}$ by $\psi(x) = \mathrm{eval}_\varphi^{\mathcal{A}}(x\mu)$ for every $x \in V$. To prove that ψ is in fact a valuation we have to show that $\mathrm{eval}_\varphi^{\mathcal{A}}(x\mu) \in A^C$ for every $x \in V^{\mathrm{C}}$. If $x \in \{x_1, \ldots, x_n\}$ then $\mathrm{eval}_\varphi^{\mathcal{A}}(x\mu) \in A^C$ by (*). If $x \in V^{\mathrm{C}} \setminus \{x_1, \ldots, x_n\}$ then $x\mu \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ so that $\mathrm{eval}_\varphi^{\mathcal{A}}(x\mu) \in A^C$ as well. Hence, ψ is a valuation. By structural induction on t one can easily show now that (**) $\mathrm{eval}_\psi^{\mathcal{A}}(t) = \mathrm{eval}_\varphi^{\mathcal{A}}(t\mu)$ for all $t \in \mathcal{T}(sig, V)$.

Because of $\mathcal{A} \models \Gamma$, $\mathcal{A}$ must satisfy a literal λ in Γ with ψ. Using (**) it is easily shown that $\mathcal{A}$ satisfies $\lambda\mu$ with φ. Hence, $\mathcal{A}$ satisfies $\Gamma\mu, \neg\mathrm{def}(x_1\mu), \ldots, \neg\mathrm{def}(x_n\mu)$ with φ. $\square$

Proof of Lemma 5.3.3. By Lemma 5.1.2, the inference rule is safe. We now prove the soundness of the rule. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

Suppose $(\langle \Gamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. If Π is not inductively valid w.r.t. *spec* then we are done (see Definition 4.1.4). Thus, let Π be inductively valid w.r.t. *spec*. By Lemma 5.3.2, the clause $\Pi\mu, \mathrm{DefCond}(\mu, \Pi)$ is also inductively valid w.r.t. *spec*. Since Γ, Θ contains $\Pi\mu, \mathrm{DefCond}(\mu, \Pi)$, Θ cannot be the empty clause because then inductive validity of $\Pi\mu, \mathrm{DefCond}(\mu, \Pi)$ would imply that of Γ. This is impossible as $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. So let Θ consist of the literals $\lambda_1, \ldots, \lambda_n$ with $n > 0$. Since σ is an inductive substitution, $(\Pi\mu, \mathrm{DefCond}(\mu, \Pi))\sigma$ is also inductively valid w.r.t. *spec* by Lemma 5.3.2. Thus, there is a literal λ in $\Pi\mu, \mathrm{DefCond}(\mu, \Pi)$ such that $\mathcal{A}$ satisfies $\lambda\sigma$ with φ. As $\mathcal{A}$ does not satisfy $\Gamma\sigma$ with φ, λ must occur in Θ; in other words there is an $i \in \{1, \ldots, n\}$ such that $\lambda =_{\mathrm{lit}} \lambda_i$. W.l.o.g. assume that i is the least number with this property, i.e. $\mathcal{A}$ does not satisfy $\lambda_j\sigma$ with φ for $j = 1, \ldots, i-1$. Definition 5.2.16 implies that $\mathcal{A}$ does not satisfy $\Lambda_i\sigma$ with φ. Hence, $(\langle \Lambda_i, \Gamma\,;\,w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample which is clearly equivalent to $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. □

Proof of Lemma 5.3.4. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

To show that the inference rule is sound assume that $(\langle \Gamma, \lambda, \Delta\,;\,w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. If $\Pi,\ l \doteq r\,, \Sigma$ is not inductively valid w.r.t. *spec* then we are done (see Definition 4.1.4). Thus, let $\Pi,\ l \doteq r\,, \Sigma$ be inductively valid w.r.t. *spec*. By Lemma 5.3.2, the clause $((\Pi,\ l \doteq r\,, \Sigma)\mu, \mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma)))\sigma$ is also inductively valid w.r.t. *spec* as σ is an inductive substitution. Hence, there is a literal λ' in $(\Pi,\ l \doteq r\,, \Sigma)\mu, \mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma))$ such that $\mathcal{A}$ satisfies $\lambda'\sigma$ with φ. Let Θ consist of the literals $\lambda_1, \ldots, \lambda_n$ for some $n \in \mathbb{N}$.

If $\mathcal{A}$ does not satisfy $\lambda_i\sigma$ with φ for $i = 1, \ldots, n$ then we have (*) $\lambda' =_{\mathrm{lit}} (\, l\mu = r\mu\,)$ since $\Gamma, \Delta,\ l\mu = r\mu\,, \Theta$ contains $(\Pi,\ l \doteq r\,, \Sigma)\mu, \mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma))$. Consequently, $\mathcal{A}$ does not satisfy $\lambda\sigma, \Gamma\sigma, \Delta\sigma, \Lambda\sigma$ with φ. Now by (*), $\mathcal{A}$ satisfies $(l\mu = r\mu)\sigma$ with φ, and because of $\lambda/p = l\mu$ we obtain that $\mathcal{A}$ does not satisfy $(\lambda[r\mu]_p)\sigma$ with φ either. Hence, $(\langle \Lambda, \Gamma, \lambda[r\mu]_p, \Delta\,;\,w\rangle, \sigma, \varphi)$ is also an $\mathcal{A}$-counterexample which is obviously equivalent to $(\langle \Gamma, \lambda, \Delta\,;\,w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$.

Otherwise, there is an $i \in \{1, \ldots, n\}$ such that $\mathcal{A}$ satisfies $\lambda_i\sigma$ with φ. W.l.o.g. assume that i is the least number having this property, i.e. $\mathcal{A}$ does not satisfy $\lambda_j\sigma$ with φ for $j = 1, \ldots, i-1$. By Definition 5.2.16, $\mathcal{A}$ does not satisfy $\Lambda_i\sigma$ with φ. Therefore, $(\langle \Lambda_i, \Gamma, \lambda, \Delta\,;\,w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample which is again equivalent to $(\langle \Gamma, \lambda, \Delta\,;\,w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$.

Let us now prove that the inference rule is safe as well. Assume that the clauses Γ, λ, Δ and $\Pi,\ l \doteq r\,, \Sigma$ are inductive theorems w.r.t. *spec*. Obviously, the clause $\Lambda_i, \Gamma, \lambda, \Delta$ is also inductively valid w.r.t. *spec* for $i = 1, \ldots, n$. Let φ be a valuation of V in $\mathcal{A}$. We still have to show that $\mathcal{A}$ satisfies $\Lambda, \Gamma, \lambda[r\mu]_p, \Delta$ with φ. If there is a literal in Γ, Δ satisfied by $\mathcal{A}$ with φ then the claim holds. Otherwise, $\mathcal{A}$ satisfies λ with φ. Furthermore, $(\Pi,\ l \doteq r\,, \Sigma)\mu, \mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma))$ is also inductively valid w.r.t. *spec*. Thus, there is a literal λ' in $(\Pi,\ l \doteq r\,, \Sigma)\mu, \mathrm{DefCond}(\mu, (\Pi,\ l \doteq r\,, \Sigma))$ such that $\mathcal{A}$ satisfies λ' with φ. If λ' occurs in Θ (and thus in Λ) then the claim is proved. Otherwise, we have $\lambda' =_{\mathrm{lit}} (\, l\mu = r\mu\,)$, i.e. $\mathcal{A}$ satisfies $l\mu = r\mu$ with φ. Since $\mathcal{A}$ satisfies λ with φ (see above) we conclude that $\mathcal{A}$ satisfies $\lambda[r\mu]_p$ with φ as well and thus $\Lambda, \Gamma, \lambda[r\mu]_p, \Delta$. □

Proof of Lemma 5.3.5. As a literal removing inference rule it is obviously sound (see the proof of Lemma 5.2.6). We still have to prove the safeness of the rule. Assume that the clauses Γ, λ, Δ and Π, λ', Σ are inductively valid w.r.t. *spec*. Let $\mathcal{A} \in \mathrm{DMod}(spec)$, and let φ be a valuation of V in $\mathcal{A}$. We have to show that $\mathcal{A}$ satisfies $\varGamma, \varDelta$ with φ. Suppose the claim is false. Then (*) $\mathcal{A}$ satisfies λ with φ. Moreover, $\mathcal{A}$ must satisfy $\lambda'\mu$ with φ since $(\varPi, \lambda', \varSigma)\mu, \mathrm{DefCond}(\mu, (\varPi, \lambda', \varSigma))$ is inductively valid w.r.t. *spec* (see Lemma 5.3.2) and since $\varGamma, \varDelta$ contains $\varPi\mu, \varSigma\mu, \mathrm{DefCond}(\mu, (\varPi, \lambda', \varSigma))$. As $\overline{\lambda} =_{\mathrm{lit}} \lambda'\mu$, $\mathcal{A}$ satisfies $\overline{\lambda}$ with φ. However, this contradicts (*). □

Proof of Lemma 5.3.6. By Lemma 5.1.2, the inference rule is safe. We now prove that the rule is sound. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

Suppose $(\langle \varGamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Let $\varTheta$ consist of the literals $\lambda_1, \ldots, \lambda_n$ for some $n \in \mathbb{N}$. If $\mathcal{A}$ does not satisfy $\lambda_i\sigma$ with φ for $i = 1, \ldots, n$ then $\mathcal{A}$ does not satisfy $\varGamma\sigma, \varTheta\sigma$ with φ. If $\mathcal{A}$ does not satisfy $(\hat{w}\mu < w)\sigma$ with φ then $(\langle\, \hat{w}\mu < w\,, \varLambda, \varGamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample, which is obvioulsy equivalent to $(\langle \varGamma\,;\, w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. So assume $\mathcal{A}$ satisfies $(\,\hat{w}\mu < w)\sigma$ with φ. Since $\mathrm{DefCond}(\mu, \varPi)$ is contained in $\varGamma, \varTheta$ there have to be variables $x_1, \ldots, x_m \in V^{\mathrm{C}} \cap \mathrm{Var}(\varGamma)$ sucht that $\mathrm{DefCond}(\mu, \varPi) = \neg\mathrm{def}(x_1\mu)\,, \ldots,\, \neg\mathrm{def}(x_m\mu)$ for some $m \in \mathbb{N}$. We define an inductive substitution τ as follows: Let $i \in \{1, \ldots, m\}$. As $\mathcal{A}$ does not satisfy $\varGamma\sigma, \varTheta\sigma$ with φ and since $\varGamma, \varTheta$ contains $\mathrm{DefCond}(\mu, \varPi)$, we obtain that $\mathcal{A}$ satisfies $\mathrm{def}(x_i\mu\sigma)$ with φ. In particular, there is a $\overline{t}_i \in \mathcal{GT}(sig^C)$ such that $\mathrm{eval}_\varphi^{\mathcal{A}}(x_i\mu\sigma) = \overline{t}_i^{\mathcal{A}}$. Define $\tau(x) = \overline{t}_i$. If $x \in V \backslash \{x_1, \ldots, x_n\}$ then define $\tau(x) = x\mu\sigma$. Since $x\mu \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ if $x \in V^{\mathrm{C}} \backslash \{x_1, \ldots, x_n\}$, we have $\tau(x) \in \mathcal{GT}(sig^C)$. Hence, τ is indeed an inductive substitution. By induction on t one can easily show now that (*) $\mathrm{eval}_\varphi^{\mathcal{A}}(t\mu\sigma) = \mathrm{eval}_\varphi^{\mathcal{A}}(t\tau)$ for every $t \in \mathcal{T}(sig, V)$. Now $\varGamma, \varTheta$ contains $\varPi\mu$ and $\mathcal{A}$ does not satisfy $\varGamma\sigma, \varTheta\sigma$ with φ. Using (*) we can infer that $\mathcal{A}$ does not satisfy $\varPi\tau$ with φ. Hence, $(\langle \varPi\,;\, \hat{w}\rangle, \tau, \varphi)$ is an $\mathcal{A}$-counterexample. We still have to show that $(\langle \varPi\,;\, \hat{w}\rangle, \tau, \varphi) \prec_{\mathcal{A}} (\langle \varGamma\,;\, w\rangle, \sigma, \varphi)$, i.e. $\mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\tau) <_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_\varphi^{\mathcal{A}}(w\sigma)$. As $\mathcal{A}$ satisfies $(\hat{w}\mu < w)\sigma$ with φ we have $\mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\mu\sigma) <_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_\varphi^{\mathcal{A}}(w\sigma)$. Now (*) implies that $\mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\mu\sigma) = \mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\tau)$ which proves the claim.

We still have to deal with the case that there is an $i \in \{1, \ldots, n\}$ such that $\mathcal{A}$ satisfies $\lambda_i\sigma$ with φ. W.l.o.g we can assume that i is the least number having this property, i.e. $\mathcal{A}$ does not satisfy $\lambda_j\sigma$ with φ for $j = 1, \ldots, i-1$. By Definition 5.2.16, $\mathcal{A}$ does not satisfy $\varLambda_i\sigma$ with φ. Thus, $(\langle \varLambda_i, \varGamma\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample, which is again equivalent to $(\langle \varGamma\,;\, w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. □

Proof of Lemma 5.3.7. Let $\mathcal{A} \in \mathrm{DMod}(spec)$.

We first prove that the inference rule is sound. Assume that $(\langle \varGamma, \lambda, \varDelta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample. Let $\varTheta$ consist of the literals $\lambda_1, \ldots, \lambda_n$ for some $n \in \mathbb{N}$.

Suppose that there is an $i \in \{1, \ldots, n\}$ such that $\mathcal{A}$ satisfies $\lambda_i\sigma$ with φ. We can assume w.l.o.g. that i is the least number with this property, i.e. $\mathcal{A}$ does not satisfy $\lambda_j\sigma$ with φ for $j = 1, \ldots, i-1$. By Definition 5.2.16, $\mathcal{A}$ does not satisfy $\varLambda_i\sigma$ with φ. Hence, $(\langle \varLambda_i, \varGamma, \lambda, \varDelta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample, which is evidently equivalent to $(\langle \varGamma, \lambda, \varDelta\,;\, w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$.

Let us now deal with the case that $\mathcal{A}$ does not satisfy $\lambda_i\sigma$ with φ for $i = 1, \ldots, n$. Then $\mathcal{A}$ does not satisfy $\varLambda\sigma$ with φ. If $\mathcal{A}$ satisfies $(l\mu = r\mu)\sigma$ with φ then $\mathcal{A}$ does not satisfy $(\lambda[r\mu]_p)\sigma$ with φ as $\mathcal{A}$ does not satisfy $\lambda\sigma$ with φ. Hence, $(\langle \varLambda, \varGamma, \lambda[r\mu]_p, \varDelta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample, which is equivalent to $(\langle \Gamma, \lambda, \Delta\,;\, w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. Otherwise, $\mathcal{A}$ does not satisfy $(l\mu = r\mu)\sigma$ with φ. Therefore, $\mathcal{A}$ does not satisfy $\Lambda'\sigma$ with φ.

If $\mathcal{A}$ does not satisfy $(\hat{w}\mu < w)\sigma$ with φ then $(\langle\, \hat{w}\mu < w\,, \Lambda', \Gamma, \lambda, \Delta\,;\, w\rangle, \sigma, \varphi)$ is an $\mathcal{A}$-counterexample, which is equivalent to $(\langle \Gamma, \lambda, \Delta\,;\, w\rangle, \sigma, \varphi)$ w.r.t. $\precsim_{\mathcal{A}}$. So assume $\mathcal{A}$ satisfies $(\hat{w}\mu < w)\sigma$ with φ. As $\mathrm{DefCond}(\mu, (\Pi,\, l \doteq r\,, \Sigma))$ is contained in $\Gamma, \Delta,\, l\mu = r\mu\,, \Theta$ there are $x_1, \ldots, x_m \in V^{\mathrm{C}} \cap \mathrm{Var}(\Pi,\, l \doteq r\,, \Sigma)$ such that $\mathrm{DefCond}(\mu, (\Pi,\, l \doteq r\,, \Sigma)) = \neg\mathrm{def}(x_1\mu)\,, \ldots,\, \neg\mathrm{def}(x_m\mu)$ for some $m \in \mathbb{N}$. We define an inductive substitution τ as follows: Let $i \in \{1, \ldots, m\}$. As $\mathcal{A}$ does not satisfy $\Gamma\sigma, \Delta\sigma, (\,l\mu = r\mu\,)\sigma, \Theta\sigma$ with φ and since $\Gamma, \Delta,\, l\mu = r\mu\,, \Theta$ contains $\mathrm{DefCond}(\mu, (\Pi,\, l \doteq r\,, \Sigma))$, we obtain that $\mathcal{A}$ satisfies $\mathrm{def}(x_i\mu\sigma)$ with φ. In particular, there is a $\bar{t}_i \in \mathcal{GT}(sig^C)$ such that $\mathrm{eval}_\varphi^{\mathcal{A}}(x_i\mu\sigma) = \bar{t}_i^{\mathcal{A}}$. Define $\tau(x) = \bar{t}_i$. If $x \in V \backslash \{x_1, \ldots, x_n\}$ then define $\tau(x) = x\mu\sigma$. Since $x\mu \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ if $x \in V^{\mathrm{C}} \backslash \{x_1, \ldots, x_n\}$, we have $\tau(x) \in \mathcal{GT}(sig^C)$. Hence, τ is indeed an inductive substitution. By induction on t one can easily show now that (*) $\mathrm{eval}_\varphi^{\mathcal{A}}(t\mu\sigma) = \mathrm{eval}_\varphi^{\mathcal{A}}(t\tau)$ for every $t \in \mathcal{T}(sig, V)$. Now $\Gamma, \Delta,\, l\mu = r\mu\,, \Theta$ contains $\Pi\mu, \Sigma\mu$ and $\mathcal{A}$ does not satisfy $\Gamma\sigma, \Delta\sigma, (l\mu = r\mu)\sigma, \Theta\sigma$ with φ. Using (*) we can infer that $\mathcal{A}$ does not satisfy $(\Pi,\, l \doteq r\,, \Sigma)\tau$ with φ. Hence, $(\langle \Pi,\, l \doteq r\,, \Sigma\,;\, \hat{w}\rangle, \tau, \varphi)$ is an $\mathcal{A}$-counterexample. We now show that $(\langle \Pi,\, l \doteq r\,, \Sigma\,;\, \hat{w}\rangle, \tau, \varphi) \prec_{\mathcal{A}} (\langle \Gamma, \lambda, \Delta\,;\, w\rangle, \sigma, \varphi)$, i.e. $\mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\tau) <_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_\varphi^{\mathcal{A}}(w\sigma)$: As $\mathcal{A}$ satisfies $(\hat{w}\mu < w)\sigma$ with φ we have $\mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\mu\sigma) <_{\mathcal{A}}^{\mathrm{lex}} \mathrm{eval}_\varphi^{\mathcal{A}}(w\sigma)$. Now (*) implies that $\mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\mu\sigma) = \mathrm{eval}_\varphi^{\mathcal{A}}(\hat{w}\tau)$ which proves the claim.

Let us now prove that the inference rule is safe. Assume that the clauses Γ, λ, Δ and $\Pi,\, l \doteq r\,, \Sigma$ are inductive theorems w.r.t. *spec*. Clearly, the clause $\Lambda_i, \Gamma, \lambda, \Delta$ is inductively valid w.r.t. *spec* for $i = 1, \ldots, n$. The same also holds for $\hat{w}\mu < w\,, \Lambda', \Gamma, \lambda, \Delta$ as it contains the clause Γ, λ, Δ. So we still have to show that $\Lambda, \Gamma, \lambda[r\mu]_p, \Delta$ is inductively valid w.r.t. *spec*. This can be done in exactly the same way as in the safeness proof for the inference rule Non-Inductive Rewriting (see the proof of Lemma 5.3.4). □

Proof of Lemma 6.1.1. We prove the lemma by induction on n.

Let $n = 0$. Since $\mu_0, \ldots, \mu_m$ is a path in G and $\nu_0 = \nu_n = \mu_0$, there is a path from ν_0 to μ_m in G.

Suppose that $n > 0$. If $\{\mu_1, \ldots, \mu_m\} \cap \{\nu_0, \ldots, \nu_{n-1}\} = \emptyset$ then $\nu_0, \ldots, \nu_n, \ldots, \mu_m$ is obviously a path from ν_0 to μ_m in G. Otherwise there is a $j_0 \in \{0, \ldots, n-1\}$ and an $i_0 \in \{1, \ldots, m\}$ such that $\nu_{j_0} = \mu_{i_0}$. Since $j_0 < n$, we can apply the induction hypothesis to $\nu_0, \ldots, \nu_{j_0}$ and $\mu_{i_0}, \ldots, \mu_m$. This gives us a path from ν_0 to μ_m in G. □

Proof of Lemma 6.1.5. Let $\nu' \in N_{\mathrm{inf}}$ such that $L((\nu', \nu)) = \mathcal{L}$. Consider a path $\nu_0, \ldots, \nu_n$ from $\tilde{\nu}$ to ν in P with $\nu_{n-1} = \nu'$. Such a path must exist (see above). By the hypothesis for ν, this path cannot be $\mathcal{L}$-free, and so there is an $i \in \{1, \ldots, n-1\}$ such that $\nu_i \in N_{\mathrm{inf}}$, $L((\nu_i, \nu_{i+1})) = \mathcal{L}$ and the path from ν_{i+1} to ν is $\mathcal{L}$-free. Suppose the goal node ν_{i+1} is not a leaf in P. Then ν_{i+1} must have been expanded in the construction of P. Hence there is another path from $\tilde{\nu}$ to ν_{i+1} in P that is $\mathcal{L}$-free (see Definition 6.1.4). This means, however, that an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν exists in P, which contradicts the hypothesis for ν. Hence, ν_{i+1} is a leaf in P so that $i + 1 = n$, ν is a leaf in P and $L((\nu_{n-1}, \nu_n)) = L((\nu', \nu)) = \mathcal{L}$. □

Proof of Lemma 6.1.7. Let $P_1 = (N, E, L) = P_2$ be a partial proof attempt both for $\tilde{\nu}_1$ and $\tilde{\nu}_2$ in G. It suffices to show that each open goal node of P_1 is also an open goal node of P_2.

Let ν be an open goal node of P_1. Then ν is a leaf in P_2. We still have to verify that there is an $\mathcal{L}$-free path from $\tilde{\nu}_2$ to ν. As ν is an open goal node of P_1 there is an $\mathcal{L}$-free path from $\tilde{\nu}_1$ to ν. If $\tilde{\nu}_1 = \nu$ then $\nu = \tilde{\nu}_2$ and there is an $\mathcal{L}$-free path from $\tilde{\nu}_2$ to ν. If $\tilde{\nu}_1 \neq \nu$ then there is a $\nu' \in N_{\mathrm{inf}}$ such that $(\nu', \nu) \in E$ and $L((\nu', \nu)) \neq \mathcal{L}$. Because of Lemma 6.1.5 we conclude that there must be an $\mathcal{L}$-free path from $\tilde{\nu}_2$ to ν. □

Proof of Lemma 6.2.3. (1) Consider the set $\mathcal{P}$ of all partial proof attempts for $\hat{\nu}$ in P. (Note that P is a proof state graph w.r.t. spec.) We define a binary relation $\longmapsto_1$ on $\mathcal{P}$ by $P_1 \longmapsto_1 P_2$ iff P_2 can be obtained from P_1 by expanding some $\nu' \in \mathrm{OGNd}(P_1)$ as in Definition 6.1.4(b). Clearly, if $P_1 \in \mathcal{P}$ then $P_2 \in \mathcal{P}$. As there are no choice-points in P, every expansion step $P_1 \longmapsto_1 P_2$ is solely determined by the choice of $\nu' \in \mathrm{OGNd}(P_1)$. Obviously, $\longmapsto_1$ is terminating. We claim that $\longmapsto_1$ is also strongly confluent (and hence confluent; see Dershowitz & Jouannaud, 1990): Given $P_1, P_2, P_3 \in \mathcal{P}$ such that $P_1 \longmapsto_1 P_2$ by expansion of some $\nu_1 \in \mathrm{OGNd}(P_1)$ and $P_1 \longmapsto_1 P_3$ by expansion of some $\nu_2 \in \mathrm{OGNd}(P_1)$, we obtain $P_2 \longmapsto_1 P_4$ and $P_3 \longmapsto_1 P_4$, where P_4 is the result of expanding ν_2 in P_2 (or ν_1 in P_3). (Note that if $\nu_1 \neq \nu_2$ then $\nu_2 \in \mathrm{OGNd}(P_2)$ and $\nu_1 \in \mathrm{OGNd}(P_3)$.) Let $\hat{P}$ be the unique normal form of $P_0 = (\{\hat{\nu}\}, \emptyset, L|_{\{\hat{\nu}\}})$ w.r.t. $\longmapsto_1$. Thus, $\hat{P}$ is a unique maximal partial proof attempt for $\hat{\nu}$ in P.

Now suppose that $\nu \in \mathrm{OGNd}(\hat{P})$. We will prove that $\nu \in \mathrm{OGNd}(P)$. Since $\hat{P}$ is $\longmapsto_1$-irreducible, ν is a leaf in P. Thus, we still have to show that (*) there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν. By the assumption for $\hat{\nu}$, there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to $\hat{\nu}$; and because of $\nu \in \mathrm{OGNd}(\hat{P})$, there is an $\mathcal{L}$-free path from $\hat{\nu}$ to ν in P. Hence, we can apply Lemma 6.1.1 to infer (*).

Finally, we have $\mathrm{LNd}(\hat{P}) \subseteq \mathrm{LNd}(P)$, as $\hat{P}$ is a subgraph of P.

(2) Let $\mathcal{P}$ be given as above. Again we define a binary relation $\longmapsto_2$ on $\mathcal{P}$ by $P_1 \longmapsto_2 P_2$ iff P_2 can be obtained from P_1 by expanding some $\nu' \in \mathrm{OGNd}(P_1)\backslash\{\tilde{\nu}\}$ as in Definition 6.1.4(b). Analogous to the proof of (1) it can easily be shown that $\longmapsto_2$ is terminating and (strongly) confluent. Let $\hat{P}$ be the unique normal form of $P_0 = (\{\hat{\nu}\}, \emptyset, L|_{\{\hat{\nu}\}})$ w.r.t. $\longmapsto_2$.

If $\hat{\nu} = \tilde{\nu}$ then $\hat{P} = P_0$ so that $\tilde{\nu} \in \mathrm{OGNd}(\hat{P})$. Otherwise there is an $\mathcal{L}$-free path $\nu_0, \ldots, \nu_n$ from $\hat{\nu}$ to $\tilde{\nu}$ in P with $n > 1$ (due to the additional assumption for $\hat{\nu}$). Thus, it is easily seen that the goal node ν_{n-2} is eventually expanded in the construction of $\hat{P}$ yielding $\tilde{\nu}$ as an open goal node of $\hat{P}$. As $\longmapsto_2$ does not allow the expansion of $\tilde{\nu}$, $\tilde{\nu}$ is a leaf in $\hat{P}$, and we conclude that $\tilde{\nu} \in \mathrm{OGNd}(\hat{P})$. That properties (ii) and (iii) hold for $\hat{P}$ can be shown just as in the proof of (1). □

Proof of Lemma 6.2.2. Let $\mathcal{A} = (A, F^{\mathcal{A}})$, and let P be a partial proof attempt for $\tilde{\nu}$. Throughout the whole proof we assume:

(*) For each $\nu \in \mathrm{LNd}(P)$ with $L(\nu) = \langle \mathit{\Pi}\,;\,\hat{w}\rangle$, $\mathit{\Pi}$ is inductively valid w.r.t. *spec*.

Hence, we will show that (2) is true for P and any $\mathcal{A}$-counterexample $(\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ by structural induction on partial proof attempts in combination with well-founded induction on the set of $\mathcal{A}$-counterexamples with respect to $\precsim_{\mathcal{A}}$ (see Section 4.1). For that purpose we define the *size* of a partial proof attempt for a goal node in G to be the number of its inference nodes.

If the size of P is 0, then $P = (\{\tilde{\nu}\}, \emptyset, L|_{\{\tilde{\nu}\}})$ and $\mathrm{OGNd}(P) = \{\tilde{\nu}\}$. Therefore, $\mathrm{IOGNd}(\tilde{\nu}, P) = \emptyset$ and (2) trivially holds for P because of $\tilde{\nu}$ and the $\mathcal{A}$-counterexample $(\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$.

Suppose now that P is of positive size. By Definition 6.1.4(b), there is a partial proof attempt $P' = (N', E', L')$ for $\tilde{\nu}$ in G and a $\nu' \in \mathrm{OGNd}(P')$ such that P can be obtained from P' by expansion of ν'. That is, there are $\nu_0, \nu_1, \ldots, \nu_n, \hat{\nu}_1, \ldots, \hat{\nu}_k \in N$ such that $(\nu, \nu_0) \in E$ and $\nu_1, \ldots, \nu_n, \hat{\nu}_1, \ldots, \hat{\nu}_k$ are exactly the successors of ν_0 in G, where $L((\nu_0, \nu_i)) = \mathcal{S}$ for $i = 1, \ldots, n$ and $L((\nu_0, \hat{\nu}_j)) \in \{\mathcal{L}, \mathcal{I}\}$ for $j = 1, \ldots, k$; and $P = (\tilde{N}, \tilde{E}, \tilde{L})$ where

- $\tilde{N} = N' \cup \{\nu_0, \nu_1, \ldots, \nu_n, \hat{\nu}_1, \ldots, \hat{\nu}_k\}$
- $\tilde{E} = E' \cup \{(\nu', \nu_0), (\nu_0, \nu_1), \ldots, (\nu_0, \nu_n), (\nu_0, \hat{\nu}_1), \ldots, (\nu_0, \hat{\nu}_k)\}$
- $\tilde{L} = L|_{\tilde{N} \cup \tilde{E}}$.

Clearly, P' is of smaller size than P. Let us apply the induction hypothesis to P' and the $\mathcal{A}$-counterexample $(\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. As $\mathrm{LNd}(P') \subseteq \mathrm{LNd}(P)$ and due to (*) we obtain a goal node $\nu \in \mathrm{OGNd}(P')$ with $L(\nu) = \langle \mathit{\Gamma}\,;\,w\rangle$ satisfying:

(3) There is an $\mathcal{A}$-counterexample of the form $(\langle \mathit{\Gamma}\,;\,w\rangle, \sigma, \varphi)$ such that

   (a) $(\langle \mathit{\Gamma}\,;\,w\rangle, \sigma, \varphi) \precsim_{\mathcal{A}} (\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ and

   (b) $\nu \in \mathrm{IOGNd}(\tilde{\nu}, P')$ implies $(\langle \mathit{\Gamma}\,;\,w\rangle, \sigma, \varphi) \prec_{\mathcal{A}} (\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$.

Case 1: $\nu \neq \nu'$

Since ν' (and not ν) was expanded to yield P, ν is a leaf also in P. As P' is a subgraph of P each path in P' is a path also in P. Thus, $\nu \in \mathrm{OGNd}(P)$. Because of (3)(a) we have $(\langle \mathit{\Gamma}\,;\,w\rangle, \sigma, \varphi) \precsim_{\mathcal{A}} (\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Suppose now that $\nu \in \mathrm{IOGNd}(\tilde{\nu}, P)$. Then there is no $\mathcal{R}/\mathcal{S}$-labeled path from $\tilde{\nu}$ to ν in P. Hence there can be no $\mathcal{R}/\mathcal{S}$-labeled path from $\tilde{\nu}$ to ν in P', and $\nu \in \mathrm{IOGNd}(\tilde{\nu}, P')$. By (3)(b), $(\langle \mathit{\Gamma}\,;\,w\rangle, \sigma, \varphi) \prec_{\mathcal{A}} (\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Therefore, (2) holds for P and $(\langle\tilde{\mathit{\Gamma}}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ because of ν and the $\mathcal{A}$-counterexample $(\langle \mathit{\Gamma}\,;\,w\rangle, \sigma, \varphi)$.

Case 2: $\nu = \nu'$

Then $L(\nu') = \langle \mathit{\Gamma}\,;\,w\rangle$. Let $L(\nu_i) = \langle \mathit{\Gamma}_i\,;\,w_i\rangle$ for $i = 1, \ldots, n$ and let $L(\hat{\nu}_j) = \langle \mathit{\Pi}_j\,;\,\hat{w}_j\rangle$ for $j = 1, \ldots, k$. From Definition 6.1.2 it follows that

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

is an instance of an inference rule. By Theorem 5.4.1, the applied inference rule is sound, and so one of the following statements must hold for the given $\mathcal{A}$-counterexample $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$ (see Definition 4.1.4):

(4) There is an $i_0 \in \{1,\ldots,n\}$ and an $\mathcal{A}$-counterexample of the form $(\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi)$ such that $(\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi) \precsim_{\mathcal{A}} (\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$.

(5) There is a $j_0 \in \{1,\ldots,k\}$ such that $U_{j_0} = \mathcal{L}$ and Π_{j_0} is not inductively valid w.r.t. *spec*.

(6) There is a $j_0 \in \{1,\ldots,k\}$ and an $\mathcal{A}$-counterexample of the form $(\langle \Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi)$ such that $U_{j_0} = \mathcal{I}$ and $(\langle \Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi) \prec_{\mathcal{A}} (\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$.

Due to (*), (5) cannot be true. Therefore, the two remaining statements (4) and (6) have to be dealt with.

Case 2.1: (4) holds.

As $\nu' \in \mathrm{OGNd}(P')$, there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν_{i_0} in P. Thus, by Lemma 6.2.3(1), there is a maximal subgraph P'' of P such that (i) P'' is a partial proof attempt for ν_{i_0} in P (ii) $\mathrm{OGNd}(P'') \subseteq \mathrm{OGNd}(P)$ and (iii) $\mathrm{LNd}(P'') \subseteq \mathrm{LNd}(P)$.

Case 2.1.1: $P'' \neq P$

Then the size of P'' is smaller than that of P, and so we can use the induction hypothesis for P'' and the $\mathcal{A}$-counterexample $(\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi)$. Since $\mathrm{LNd}(P'') \subseteq \mathrm{LNd}(P)$ and because of (*) there must be a $\nu'' \in \mathrm{OGNd}(P'')$ with $L(\nu'') = \langle \Gamma''\,;\,w''\rangle$ such that:

(7) There is an $\mathcal{A}$-counterexample of the form $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'')$ such that

  (a) $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \precsim_{\mathcal{A}} (\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi)$ and

  (b) $\nu'' \in \mathrm{IOGNd}(\nu_{i_0}, P'')$ implies $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \prec_{\mathcal{A}} (\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi)$.

Now $\nu'' \in \mathrm{OGNd}(P)$, as $\mathrm{OGNd}(P'') \subseteq \mathrm{OGNd}(P)$ by Lemma 6.2.3(1). Due to (7)(a) and (4) we have $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \precsim_{\mathcal{A}} (\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi) \precsim_{\mathcal{A}} (\langle \Gamma\,;\,w\rangle, \sigma, \varphi)$. Since $\nu' = \nu$, it follows that $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi) \precsim_{\mathcal{A}} (\langle \tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ by (3)(a). As a consequence, $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \precsim_{\mathcal{A}} (\langle \tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$.

We still have to show that $\nu'' \in \mathrm{IOGNd}(\tilde{\nu}, P)$ implies that $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \prec_{\mathcal{A}} (\langle \tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. If $\nu'' \notin \mathrm{IOGNd}(\nu_{i_0}, P'')$ and $\nu' \notin \mathrm{IOGNd}(\tilde{\nu}, P')$ then there is an $\mathcal{R}/\mathcal{S}$-labeled path from ν_{i_0} to ν'' in P'' and an $\mathcal{R}/\mathcal{S}$-labeled path from $\tilde{\nu}$ to ν' in P'. As a proof state graph becomes an acyclic graph when the arcs labeled with $\mathcal{L}$ or $\mathcal{I}$ are removed, there is an $\mathcal{R}/\mathcal{S}$-labeled path from $\tilde{\nu}$ to ν'' in P so that $\nu'' \notin \mathrm{IOGNd}(\tilde{\nu}, P)$. Therefore, if $\nu'' \in \mathrm{IOGNd}(\tilde{\nu}, P)$ then $\nu'' \in \mathrm{IOGNd}(\nu_{i_0}, P'')$ or $\nu' \in \mathrm{IOGNd}(\tilde{\nu}, P')$. With (7)(b) and (3)(b) we can infer that $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \prec_{\mathcal{A}} (\langle \Gamma_{i_0}\,;\,w_{i_0}\rangle, \tau, \psi)$ or $(\langle \Gamma\,;\,w\rangle, \sigma, \varphi) \prec_{\mathcal{A}} (\langle \tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Hence, by (4), we have $(\langle \Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \prec_{\mathcal{A}} (\langle \tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ given that $\nu'' \in \mathrm{IOGNd}(\tilde{\nu}, P)$.

We conclude that (2) holds for P and $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$ because of ν'' and the $\mathcal{A}$-counterexample $(\langle\Gamma''\,;\,w''\rangle,\sigma'',\varphi'')$.

Case 2.1.2: $P'' = P$

Clearly, there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν_{i_0} in P (see above). Now $P'' = P$ implies that $\tilde{\nu} \notin \mathrm{OGNd}(P'')$. Consequently, $\tilde{\nu}$ must have been expanded in the construction of P'', and it follows that there is also an $\mathcal{L}$-free path from ν_{i_0} to $\tilde{\nu}$ in P. Hence, by Lemma 6.2.3(2), we obtain a subgraph $\hat{P}$ of P such that (i) $\hat{P}$ is a partial proof attempt for ν_{i_0} in P (ii) $\tilde{\nu} \in \mathrm{OGNd}(\hat{P})$ (iii) $\mathrm{OGNd}(\hat{P})\backslash\{\tilde{\nu}\} \subseteq \mathrm{OGNd}(P)$ and (iv) $\mathrm{LNd}(\hat{P}) \subseteq \mathrm{LNd}(P)$.

As $\tilde{\nu}\in\mathrm{OGNd}(\hat{P})$, the size of $\hat{P}$ must be smaller than that of P, and we can make use of the induction hypothesis for $\hat{P}$ and the $\mathcal{A}$-counterexample $(\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi)$. Since $\mathrm{LNd}(\hat{P}) \subseteq \mathrm{LNd}(P)$ and because of (*), there must be a $\hat{\nu}\in\mathrm{OGNd}(\hat{P})$ with $L(\hat{\nu}) = \langle\hat{\Gamma}\,;\,\hat{w}\rangle$ such that:

(8) There is an $\mathcal{A}$-counterexample of the form $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi})$ such that

- (a) $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi}) \precsim_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi)$
- (b) $\hat{\nu}\in\mathrm{IOGNd}(\nu_{i_0},\hat{P})$ implies $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi}) \prec_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi)$.

Case 2.1.2.1: $\hat{\nu} \neq \tilde{\nu}$

Then $\hat{\nu}\in\mathrm{OGNd}(P)$, as $\mathrm{OGNd}(\hat{P})\backslash\{\tilde{\nu}\} \subseteq \mathrm{OGNd}(P)$ by Lemma 6.2.3(2). By (8)(a) and (4), $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi}) \precsim_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi) \precsim_{\mathcal{A}} (\langle\Gamma\,;\,w\rangle,\sigma,\varphi)$. Since $\nu' = \nu$, it follows that $(\langle\Gamma\,;\,w\rangle,\sigma,\varphi) \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$ by (3)(a). Hence, $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi}) \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$.

Now let $\hat{\nu}\in\mathrm{IOGNd}(\tilde{\nu},P)$. As in Case 2.1.1 one shows that then $\hat{\nu}\in\mathrm{IOGNd}(\nu_{i_0},\hat{P})$ or $\nu'\in\mathrm{IOGNd}(\tilde{\nu},P')$. Thus, $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi}) \prec_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi)$ or $(\langle\Gamma\,;\,w\rangle,\sigma,\varphi) \prec_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$ with (8)(b) and (3)(b). By (4) we get $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi}) \prec_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$.

Again we conclude that (2) holds for P and $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$ because of $\hat{\nu}$ and the $\mathcal{A}$-counterexample $(\langle\hat{\Gamma}\,;\,\hat{w}\rangle,\hat{\sigma},\hat{\varphi})$.

Case 2.1.2.2: $\hat{\nu} = \tilde{\nu}$

Then (8) can be instantiated with $\tilde{\nu}$ for $\hat{\nu}$. That is:

(9) There is an $\mathcal{A}$-counterexample of the form $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\tau},\tilde{\psi})$ such that

- (a) $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\tau},\tilde{\psi}) \precsim_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi)$ and
- (b) $\tilde{\nu}\in\mathrm{IOGNd}(\nu_{i_0},\hat{P})$ implies $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\tau},\tilde{\psi}) \prec_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi)$.

Due to (9)(a) and (4) we have $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\tau},\tilde{\psi}) \precsim_{\mathcal{A}} (\langle\Gamma_{i_0}\,;\,w_{i_0}\rangle,\tau,\psi) \precsim_{\mathcal{A}} (\langle\Gamma\,;\,w\rangle,\sigma,\varphi)$. With $\nu' = \nu$ and (3)(a) we get $(\langle\Gamma\,;\,w\rangle,\sigma,\varphi) \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$. Thus, we obtain $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\tau},\tilde{\psi}) \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle,\tilde{\sigma},\tilde{\varphi})$.

Since every proof state graph whose arcs labeled with $\mathcal{L}$ or $\mathcal{I}$ have been removed is an acyclic graph, the directed cycle $\tilde{\nu},\ldots,\nu',\nu_0,\nu_{i_0},\ldots,\tilde{\nu}$ in P is $\mathcal{L}$-free, and (at least) one of its arcs must be labeled with $\mathcal{I}$. As a consequence, there is no $\mathcal{R}/\mathcal{S}$-labeled path from ν_{i_0} to $\tilde{\nu}$ in $\hat{P}$ or there is no $\mathcal{R}/\mathcal{S}$-labeled path from $\tilde{\nu}$ to ν' in P'. Thus, $\tilde{\nu} \in \mathrm{IOGNd}(\nu_{i_0}, \hat{P})$ or $\nu' \in \mathrm{IOGNd}(\tilde{\nu}, P')$. By (9)(b) and (3)(b), $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\tau}, \tilde{\psi}) \prec_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Hence, we can apply the induction hypothesis to P and $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\tau}, \tilde{\psi})$. Due to (*), there must be a $\nu'' \in \mathrm{OGNd}(P)$ with $L(\nu'') = \langle\Gamma''\,;\,w''\rangle$ such that:

(10) There is an $\mathcal{A}$-counterexample of the form $(\langle\Gamma''\,;\,w''\rangle, \tau'', \psi'')$ such that

$$(\langle\Gamma''\,;\,w''\rangle, \tau'', \psi'') \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\tau}, \tilde{\psi}).$$

With (10) we get $(\langle\Gamma''\,;\,w''\rangle, \tau'', \psi'') \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\tau}, \tilde{\psi}) \prec_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Hence, (2) holds for P and $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ due to ν'' and the $\mathcal{A}$-counterexample $(\langle\Gamma''\,;\,w''\rangle, \tau'', \psi'')$.

Case 2.2: (6) holds.

As $\nu' \in \mathrm{OGNd}(P')$, there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to $\hat{\nu}_{j_0}$ in P. Thus, by Lemma 6.2.3(1), there is a subgraph P'' of P such that (i) P'' is a partial proof attempt for $\hat{\nu}_{j_0}$ in P (ii) $\mathrm{OGNd}(P'') \subseteq \mathrm{OGNd}(P)$ and (iii) $\mathrm{LNd}(P'') \subseteq \mathrm{LNd}(P)$. Due to (6) and (3) (recall that $\nu = \nu'$) we have $(\langle\Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi) \prec_{\mathcal{A}} (\langle\Gamma\,;\,w\rangle, \sigma, \varphi) \precsim_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Hence, $(\langle\Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi) \prec_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$.

Thus, we can make use of the induction hypothesis for P'' and $(\langle\Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi)$, regardless of whether the size of P'' is smaller than that of P. As $\mathrm{LNd}(P'') \subseteq \mathrm{LNd}(P)$ and because of (*), there must be a $\nu'' \in \mathrm{OGNd}(P'')$ with $L(\nu'') = \langle\Gamma''\,;\,w''\rangle$ such that:

(11) There is an $\mathcal{A}$-counterexample of the form $(\langle\Gamma''\,;\,w''\rangle, \sigma'', \varphi'')$ such that

$$(\langle\Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \precsim_{\mathcal{A}} (\langle\Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi).$$

Now $\nu'' \in \mathrm{OGNd}(P)$, as $\mathrm{OGNd}(P'') \subseteq \mathrm{OGNd}(P)$ by Lemma 6.2.3(1). Due to (11) we obtain $(\langle\Gamma''\,;\,w''\rangle, \sigma'', \varphi'') \precsim_{\mathcal{A}} (\langle\Pi_{j_0}\,;\,\hat{w}_{j_0}\rangle, \tau, \psi) \prec_{\mathcal{A}} (\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$.

Again we have shown that (2) holds for P and $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$ – in this final sub-case because of ν'' and the $\mathcal{A}$-counterexample $(\langle\Gamma''\,;\,w''\rangle, \sigma'', \varphi'')$. □

Proof of Theorem 6.2.4. (1) Assume that $\tilde{\Gamma}$ is not inductively valid w.r.t. *spec*. Then there is a data model $\mathcal{A}$ of *spec* such that $\mathcal{A} \not\models \tilde{\Gamma}$. By Lemma 4.1.1, there must be an $\mathcal{A}$-counterexample of the form $(\langle\tilde{\Gamma}\,;\,\tilde{w}\rangle, \tilde{\sigma}, \tilde{\varphi})$. Since P is a proof graph, we have that, for each $\nu \in \mathrm{LNd}(P)$ with $L(\nu) = \langle\Pi\,;\,\hat{w}\rangle$, Π is inductively valid w.r.t. *spec*. Hence, by Lemma 6.2.2(2), there must be a $\nu \in \mathrm{OGNd}(P)$ with the properties stated there. However, $\mathrm{OGNd}(P) = \emptyset$ as P is a proof graph. Having derived a contradiction from our assumption we conclude that $\tilde{\Gamma}$ is inductively valid w.r.t. *spec*.

(2) If there is no $\mathcal{L}$-free path from $\tilde{\nu}$ to ν in P then $\nu \in \mathrm{LNd}(P)$, as is implied by Lemma 6.1.5. Since P is a proof graph, Γ is inductively valid w.r.t. *spec*. So assume there is an $\mathcal{L}$-free path from $\tilde{\nu}$ to ν in P. By Lemma 6.2.3(1), a partial proof attempt $\hat{P}$ for ν in P exists such that $\mathrm{OGNd}(\hat{P}) \subseteq \mathrm{OGNd}(P) = \emptyset$ and $\mathrm{LNd}(\hat{P}) \subseteq \mathrm{LNd}(P)$. It is clear that $\hat{P}$ is a proof graph for ν in G. Now one can show exactly as in the proof of (1) that Γ is inductively valid w.r.t. *spec*. □

Proof of Theorem 6.2.5. The theorem can be easily proved by structural induction on the proof state graph G. Essentially, the validity of the induction step follows from the fact that — by Theorem 5.4.1 — each inference rule applied in the construction of G is *safe* (see Definition 5.1.1). $\square$

Appendix B

Syntax of the Command Language of QuodLibet

This appendix contains a complete definition of the syntax of the command language of QuodLibet in Extended Backus-Naur Form (EBNF). Every EBNF production consists of a non-terminal symbol and an EBNF expression seperated by '::=' and terminated with a period. For each non-terminal symbol *N* there is exactly one EBNF production of the form

N ::= *EBNF-Expression* .

where *EBNF-Expression* is the definition of *N*. We use the following notation for EBNF expressions:

Notation	**Meaning**
|	alternatively
[X]	0 or 1 instance of X
{ X }*	0 or more instances of X
{ X }+	1 or more instances of X
(X | Y)	a grouping: either X or Y
"`XYZ`"	the terminal symbol XYZ

The start symbol for the syntax of the command language is *Command*. Note that the command interpreter accepts both small and capital letters in terminal symbols.

Overview of the QuodLibet Commands

Command ::= *Initialize* | *Declare* | *Define* | *Assert* |
Prove | *Assume* | *Apply* | *AssignName* |
Delete | *Set* | *Reset* | *Display* |
Compile | *Load* | *Unload* | *Call* |
Execute | *Save* | *Restore* | *Quit* .

`initialize` Command

Initialize ::= “`initialize`” .

`declare` Command

Declare ::= “`declare`” *DeclaredObject* .

DeclaredObject ::= “`sorts`” { *Sort-Ident* }$^+$ “`.`” |
“`constructor`” “`variables`” { *VarDeclarations* }$^+$ “`.`” |
“`general`” “`variables`” { *VarDeclarations* }$^+$ “`.`” |
“`operators`” { *OpDeclarations* }$^+$ “`.`” .

VarDeclarations ::= *Variable-Ident* { “`,`” *Variable-Ident* }* “`:`” *Sort-Ident* .

OpDeclarations ::= *FunctionSymbol* { “`,`” *FunctionSymbol* }* “`:`”
{ *Sort-Ident* }* “`-->`” *Sort-Ident* .

`define` Command

Define ::= “`define`” *DefinedObject* .

DefinedObject ::= “`sort`” *Sort-Ident* “`with`” “`constructors`”
{ *OpDeclarations* }$^+$ “`.`” |
“`operator`” *FunctionSymbol* “`:`” { *Sort-Ident* }* “`-->`” *Sort-Ident*
“`with`” “`defining`” “`rules`” { *RuleDefinition* }$^+$ “`.`” .

RuleDefinition ::= *DefRule-Ident* “`:`” *DefRule* .

`assert` Command

Assert ::= “`assert`” { *RuleDefinition* }$^+$ “`.`” .

`prove` Command

Prove ::= “`prove`” *Formula PS-Tree-Ident* .

`assume` Command

Assume ::= “`assume`” *Formula PS-Tree-Ident* .

`apply` Command

Apply ::= “`apply`” *InferenceRule* [*PS-Tree-Ident*] [*GNode-Position*] .

InferenceRule ::= “`compl-lit`” *LiteralNumber LiteralNumber* |
“`=/=`” “`-`” “`taut`” *LiteralNumber* |
“`<`” “`-`” “`taut`” *LiteralNumber* |
“`=`” “`-`” “`decomp`” *LiteralNumber* |
“`def`” “`-`” “`decomp`” *LiteralNumber* |
“`<`” “`-`” “`decomp`” *LiteralNumber* |
“`mult-lit`” *LiteralNumber LiteralNumber* |
“`=`” “`-`” “`removal`” *LiteralNumber* |
“`<`” “`-`” “`removal`” *LiteralNumber* |
“`=/=`” “`-`” “`removal`” *LiteralNumber* |
“`~`” “`def`” “`-`” “`removal`” *LiteralNumber* |
“`const-rewrite`” *LiteralNumber LiteralNumber Literal-Position* |
“`=/=`” “`-`” “`unif`” *LiteralNumber* |
“`ctr-var-add`” *LiteralNumber Term* |
“`tuple`” “`<`” “`-`” “`reduct`” *LiteralNumber* |
“`tuple`” “`=`” “`-`” “`reduct`” *LiteralNumber* |
“`<`” “`-`” “`mono`” *LiteralNumber Term-Position Term-Position* |
“`<`” “`-`” “`trans`” *LiteralNumber Weight* |
“`subst-add`” $\{$ *Substitution* $\}^+$ “`.`” |
“`lit-add`” $\{$ *Literal* $\}^+$ “`.`” |
“`axiom-subs`” *DefRule-Ident Substitution* |
“`lemma-subs`” *PS-Tree-Ident Substitution* |
“`axiom-rewrite`” *LiteralNumber Literal-Position DefRule-Ident*
    LiteralNumber Substitution |
“`lemma-rewrite`” *LiteralNumber Literal-Position PS-Tree-Ident*
    LiteralNumber Substitution |
“`appl-lit-removal`” *LiteralNumber Substitution*
    (*DefRule-Ident* | *PS-Tree-Ident*) *LiteralNumber* |
“`ind-subs`” *PS-Tree-Ident Substitution* |
“`ind-rewrite`” *LiteralNumber Literal-Position PS-Tree-Ident*
    LiteralNumber Substitution .

`assign-name` Command

AssignName ::= “`assign-name`” *PS-Tree-Ident* [*PS-Tree-Ident*] [*GNode-Position*] .

`delete` Command

Delete ::= “`delete`” *DeletedObject* .

DeletedObject ::= “`defining`” “`rule`” *DefRule-Ident* |
“`PS-tree`” [*PS-Tree-Ident*] |
“`I-node`” [*PS-Tree-Ident*] *INode-Position* .

`set` Command

Set ::= “`set`” *SetObject* .

SetObject ::= “`current`” “`PS-tree`” *PS-Tree-Ident* |
“`current`” “`G-node`” [*PS-Tree-Ident*] *GNode-Position* |
“`weight`” *Weight* [*PS-Tree-Ident*] |
“`search-strategy`” (“`depth-first`” | “`breadth-first`” | “`none`”)
“`display-mode`” (“`normal`” | “`detailed`”)
“`script-directory`” *Directory* |
“`qml-directory`” *Directory* .

Directory ::= *String* .

`reset` Command

Reset ::= “`reset`” *ResetObject* .

ResetObject ::= “`weight`” [*PS-Tree-Ident*] |
“`script-directory`” | “`qml-directory`” .

`display` Command

Display ::= “`display`” *DisplayedObject* .

DisplayedObject ::= [“`constructor`” | “`general`”] “`variables`” [*Sort-Ident*] |
“`signature`” |
“`defining`” “`rules`” [*FunctionSymbol*] |
“`specification`” |
“`G-node`” [*PS-Tree-Ident*] [*GNode-Position*] |
“`PS-tree-name`” |
“`PS-tree`” [*PS-Tree-Ident*] [“`I-nodes`”] |
“`conjecture`” [*PS-Tree-Ident*] |
[“`proved`” | “`unproved`”] (“`PS-tree-names`” | “`conjectures`”) .

`compile` Command

Compile ::= “`compile`” *QML-File* .

QML-File ::= *String* .

`load` Command

Load ::= "`load`" *QML-File* .

`unload` Command

Unload ::= "`unload`" *QML-File* .

`call` Command

Call ::= "`call`" *CalledObject* .

CalledObject ::= *Tactic-Ident* { *Argument* }* [*PS-Tree-Ident*] [*GNode-Position*] |
Procedure-Ident { *Argument* }* .

Argument ::= *Integer* | *Real* | *String* | *Boolean* |
Sort-Ident | *FunctionSymbol* | *Term* | *Literal* |
Formula | *Substitution* | *Position* |
Weight | *DNode* | *GNode* | *INode* .

`execute` Command

Execute ::= "`execute`" *CommandScript* ["`echo`"] .

CommandScript ::= *String*

`save` Command

Save ::= "`save`" "`IM-state`" *StateFile* .

StateFile ::= *String*

`restore` Command

Restore ::= "`restore`" "`IM-state`" *StateFile* .

`quit` Command

Quit ::= "`quit`" .

Command Parameters

Formula ::= “{” [*Literal* { “,” *Literal* }*] “}” .

DefRule ::= *Term* “`=`” *Term* [“`if`” *Literal* { “,” *Literal* }*] .

Literal ::= *Term* “`=`” *Term* | *Term* “`=/=`” *Term* |
“`def`” *Term* | “∼” “`def`” *Term* |
Weight “`<`” *Weight* | “∼” “(” *Weight* “`<`” *Weight* “)” .

Weight ::= *Term* | “(” [*Term* { “,” *Term* }*] “)” .

Substitution ::= “`[`” [*Variable-Ident* “`<--`” *Term*
{ “,” *Variable-Ident* “`<--`” *Term* }*] “`]`” .

Term ::= *Variable-Ident* | *Nat* | *Ident* [“(” *Term* { “,” *Term* }* “)”] .

DNode ::= (*DefRule-Ident* | *PS-Tree-Ident*) .

GNode ::= [*PS-Tree-Ident*] *GNode-Position* .

INode ::= [*PS-Tree-Ident*] *INode-Position* .

GNode-Position ::= *Position* .

INode-Position ::= *Position* .

Literal-Position ::= *Position* .

Term-Position ::= *Position* .

Position ::= “`root`” | “`[`” *Nat* [“`^`” *Nat*] { “`:`” *Nat* [“`^`” *Nat*] }* “`]`” .

Sort-Ident ::= *Ident* .

Variable-Ident ::= *Ident* .

DefRule-Ident ::= *Ident* .

PS-Tree-Ident ::= *Ident* .

Tactic-Ident ::= *Ident* .

Procedure-Ident ::= *Ident* .

LiteralNumber ::= *Nat* .

FunctionSymbol ::= *Ident* | *Nat* .

Token

Ident ::= *Letter* { *Letter* | *Digit* | *SpecialCharacter* $\}^*$.

String ::= “`"`” { *StringCharacter* | “`" "`” $\}^*$ “`"`” .

Boolean ::= “`true`” | “`false`” .

Integer ::= [“`+`” | “`-`”] *Nat* .

Real ::= [“`+`” | “`-`”] (*PositiveReal* | *Nat*) .

Nat ::= { *Digit* $\}^+$.

PositiveReal ::= { *Digit* $\}^+$ “`.`” { *Digit* $\}^+$ [*ScaleFactor*] |
{ *Digit* $\}^+$ *ScaleFactor* .

ScaleFactor ::= (“`e`” | “`e`”) [“`+`” | “`-`”] { *Digit* $\}^+$.

Letter ::= “`a`” | ... | “`z`” | “`A`” | ... | “`Z`” .

Digit ::= “`0`” | ... | “`9`” .

SpecialCharacter ::= “`-`” | “`_`” .

StringCharacter ::= All symbols with ASCII-code greater than 31
except for the symbol “`"`” .

Appendix C

The Inference Rules of QuodLibet

This appendix contains a complete compilation of the inference rules of QuodLibet. The following presentation of the inference rules differs from that of Chapter 5 in that, for each inference rule, we additionally indicate (i) the name of the inference rule as it is expected by the `apply` command and (ii) the *parameters* of the inference rule (see Section 7.3.2 and Appendix B). The purpose of the parameters of an inference rule is to precisely characterize actual applications of the given inference rule.

To give an example, let us consider how the inference rule Constant Rewriting (see Figure 5.6) is specified in this appendix:

Constant Rewriting:

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

$$\frac{\langle \Gamma, \lambda[t_1]_p, \Delta\,;\, w\rangle}{\langle \Gamma, \lambda[t_2]_p, \Delta\,;\, w\rangle}$$

if
- $(\Gamma, \lambda, \Delta)[m] = \lambda$
- $p \in \mathrm{Pos}(\lambda)$ and $\lambda/p = t_1$
- there is a literal $t_1 \dot{\neq} t_2$ in Γ, Δ such that $(\Gamma, \lambda, \Delta)[n] = t_1 \dot{\neq} t_2$.

This presentation of Constant Rewriting includes, besides the information given in the original definition in Figure 5.6, the string `const-rewrite` that serves as the name of the inference rule in the command language, as well as the parameters m, n and p. Moreover, observe the extended applicability conditions presented here, which also define the parameters m and n.

The reader is asked to refer to Chapter 5 for most of the notations used in this appendix. As in Chapter 5, we use the following (meta-) variables without further explanation: (i) m, n, k for natural numbers (ii) t, u, v, l, r for terms (iii) p for positions of terms in literals or terms (iv) w for weights (v) λ for literals and (vi) $\Gamma, \Delta, \Pi, \Lambda, \Sigma$ for clauses, i.e. (possibly empty) sequences of literals.

Note that $\Gamma[m]$ is to denote the m-th literal in a clause Γ. For instance, the expression $(\Gamma, \lambda, \Delta)[n] = t_1 \dot{\neq} t_2$ means that the n-th literal in the clause Γ, λ, Δ is $t_1 \dot{\neq} t_2$.

C.1 Non-Applicative Inference Rules

C.1.1 Establishing Simple Tautologies

Complementary Literals:

`compl-lit` m n

$$\frac{\langle \Gamma, \lambda, \Delta, \lambda', \Pi \,;\, w\rangle}{} \qquad \text{if} \quad \begin{array}{l} - \ (\Gamma, \lambda, \Delta, \lambda', \Pi)[m] = \lambda \\ - \ (\Gamma, \lambda, \Delta, \lambda', \Pi)[n] = \lambda' \\ - \ \lambda' =_{\text{lit}} \overline{\lambda}. \end{array}$$

$\neq$-Tautology:

`=/=-taut` m

$$\frac{\langle \Gamma,\, t_1 \neq t_2 \,, \Delta \,;\, w\rangle}{} \qquad \text{if} \quad \begin{array}{l} - \ (\Gamma,\, t_1 \neq t_2 \,, \Delta)[m] = \ t_1 \neq t_2 \\ - \ t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}}) \\ - \ t_1 \text{ and } t_2 \text{ are not unifiable.} \end{array}$$

$<$-Tautology:

`<-taut` m

$$\frac{\langle \Gamma,\, () < (u_1, \ldots, u_k) \,, \Delta \,;\, w\rangle}{} \qquad \text{if} \quad \begin{array}{l} - \ (\Gamma,\, () < (u_1, \ldots, u_k) \,, \Delta)[m] = \ () < (u_1, \ldots, u_k) \\ - \ k > 0. \end{array}$$

C.1.2 Decomposing Atoms

$=$-Decomposition:

`=-decomp` m

$$\frac{\langle \Gamma,\, t_1 = t_2 \,, \Delta \,;\, w\rangle}{\langle\, u_1 = v_1 \,, \Gamma,\, t_1 = t_2 \,, \Delta \,;\, w\rangle \ \ \ldots \ \ \langle\, u_k = v_k \,, \Gamma,\, t_1 = t_2 \,, \Delta \,;\, w\rangle}$$

$$\text{if} \quad \begin{array}{l} - \ (\Gamma,\, t_1 = t_2 \,, \Delta)[m] = \ t_1 = t_2 \\ - \ \operatorname{top}(t_1) = \operatorname{top}(t_2) \\ - \ u_1 = v_1 \,, \ldots,\, u_k = v_k \ \text{ are exactly those equations in} \\ \qquad \{\, t_1/p \doteq t_2/p \mid p \in \operatorname{MinDifPos}(t_1, t_2) \,\} \\ \quad \text{whose complements do not occur in } \Gamma, \Delta. \end{array}$$

def-Decomposition:

`def-decomp` m

$$\frac{\langle \Gamma,\ \mathrm{def}(t)\ ,\Delta\ ;\ w\rangle}{\langle\ \mathrm{def}(u_1)\ ,\Gamma,\ \mathrm{def}(t)\ ,\Delta\ ;\ w\rangle\ \ \ldots\ \ \langle\ \mathrm{def}(u_k)\ ,\Gamma,\ \mathrm{def}(t)\ ,\Delta\ ;\ w\rangle}$$

if
- $(\Gamma,\ \mathrm{def}(t)\ ,\Delta)[m] = \ \mathrm{def}(t)$
- $\mathrm{top}(t) \in (C \cup V^{\mathrm{C}})$
- $\mathrm{def}(u_1)\ ,\ldots,\ \mathrm{def}(u_k)$ are exactly those definedness atoms in
  $\{\ \mathrm{def}(t/p)\ \mid p \in \mathrm{MinNonCPos}(t)\ \}$
  whose complements do not occur in Γ, Δ.

<-Decomposition:

`<-decomp` m

$$\frac{\langle \Gamma,\ t_1 < t_2\ ,\Delta\ ;\ w\rangle}{\langle\ \mathrm{def}(u_1)\ ,\Gamma,\ t_1 < t_2\ ,\Delta\ ;\ w\rangle\ \ \ldots\ \ \langle\ \mathrm{def}(u_k)\ ,\Gamma,\ t_1 < t_2\ ,\Delta\ ;\ w\rangle}$$

if
- $(\Gamma,\ t_1 < t_2\ ,\Delta)[m] = \ t_1 < t_2$
- $\mathrm{top}(t_2) \in C$
- there are $\hat{t}_1, \hat{t}_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ 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 $x \in V^{\mathrm{C}}$
  - $\mathrm{def}(u_1)\ ,\ldots,\ \mathrm{def}(u_k)$ are exactly those definedness atoms in
    $\{\ \mathrm{def}(t_i/p) \mid i \in \{1,2\} \wedge p \in \mathrm{MinNonCPos}(t_i)\ \}$
    that do not occur in $\overline{\Gamma}, \overline{\Delta}$.

C.1.3 Removing Redundant Literals

Multiple Literals:

`mult-lit` $m\ n$

$$\frac{\langle \Gamma, \lambda, \Delta, \lambda', \Pi\ ;\ w\rangle}{\langle \Gamma, \lambda, \Delta, \Pi\ ;\ w\rangle}$$

if
- $(\Gamma, \lambda, \Delta, \lambda', \Pi)[m] = \lambda$
- $(\Gamma, \lambda, \Delta, \lambda', \Pi)[n] = \lambda'$
- $\lambda' =_{\mathrm{lit}} \lambda$.

=-Removal:

`=-removal` m

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

if
- $(\Gamma,\, t_1 = t_2\,,\Delta)[m] = \, t_1 = t_2$
- $t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}})$
- t_1 and t_2 are not unifiable.

<-Removal:

`<-removal` m

$$\frac{\langle \Gamma,\, (u_1,\ldots,u_k) < ()\,,\Delta\,;\,w\rangle}{\langle \Gamma,\Delta\,;\,w\rangle}$$

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

≠-Removal:

`=/=-removal` m

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

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

¬def-Removal:

`~def-removal` m

$$\frac{\langle \Gamma,\, \neg\mathrm{def}(t)\,,\Delta\,;\,w\rangle}{\langle \Gamma,\Delta\,;\,w\rangle}$$

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

C.1.4 Making Use of Negative Literals

Constant Rewriting:

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

$$\frac{\langle \Gamma, \lambda[t_1]_p, \Delta\,;\,w\rangle}{\langle \Gamma, \lambda[t_2]_p, \Delta\,;\,w\rangle}$$

if
- $(\Gamma,\lambda,\Delta)[m] = \lambda$
- $p \in \mathrm{Pos}(\lambda)$ and $\lambda/p = t_1$
- there is a literal $t_1 \dot{\neq} t_2$ in Γ,Δ such that $(\Gamma,\lambda,\Delta)[n] = \, t_1 \dot{\neq} t_2\,.$

$\neq$-Unification:

`=/=-unif` m

$$\frac{\langle \Gamma,\ t_1 \neq t_2\,,\, \Delta\,;\ w\rangle}{\langle \Gamma\tau, \Delta\tau\,;\ w\tau\rangle} \qquad \text{if} \quad \begin{array}{l} -\ \ (\Gamma,\ t_1 \neq t_2\,,\, \Delta)[m] =\ t_1 \neq t_2 \\ -\ \ t_1, t_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}}) \\ -\ \ \tau = \mathrm{mgu}(\,t_1 = t_2\,, \mathrm{Var}(\Gamma, t_1, t_2, \Delta, w))\ \text{exists.} \end{array}$$

Constructor Variable Addition:

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

$$\frac{\langle \Gamma,\ \neg\mathrm{def}(t)\,,\, \Delta\,;\ w\rangle}{\langle \Gamma,\ x \neq t\,,\, \Delta\,;\ w\rangle} \qquad \text{if} \quad \begin{array}{l} -\ \ (\Gamma,\ \neg\mathrm{def}(t)\,,\, \Delta)[m] =\ \neg\mathrm{def}(t) \\ -\ \ x \in V^{\mathrm{C}} \backslash \mathrm{Var}(\Gamma, \Delta, t, w). \end{array}$$

C.1.5 Further Inference Rules for Order Atoms

Tuple $<$-Reduction:

`tuple <-reduct` k

$$\frac{\langle \Gamma,\ (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,,\, \Delta\,;\ w\rangle}{\langle\, t_1 < t_2\,, \Gamma,\ (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,,\, \Delta\,;\ w\rangle}$$

$$\text{if} \quad \begin{array}{l} -\ \ (\Gamma, (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n)\,,\, \Delta)[k] =\ (t_1, u_1, \ldots, u_m) < (t_2, v_1, \ldots, v_n) \\ -\ \ m > 0\ \vee\ n > 0. \end{array}$$

Tuple $=$-Reduction:

`tuple =-reduct` k

$$\frac{\langle \Gamma,\ (t, u_1, \ldots, u_m) < (t, v_1, \ldots, v_n)\,,\, \Delta\,;\ w\rangle}{\langle \Gamma,\ (u_1, \ldots, u_m) < (v_1, \ldots, v_n)\,,\, \Delta\,;\ w\rangle}$$

$$\text{if} \quad (\Gamma, (t, u_1, \ldots, u_m) < (t, v_1, \ldots, v_n)\,,\, \Delta)[k] =\ (t, u_1, \ldots, u_m) < (t, v_1, \ldots, v_n)\,.$$

<-Monotonicity:

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

$$\frac{\langle \mathit{\Gamma},\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,,\mathit{\Delta}\ ;\ w\rangle}{\langle\, u_1 < u_2\,, \mathit{\Gamma},\ t_1[u_1]_{p_1} < t_2[u_2]_{p_2}\,, \mathit{\Delta}\ ;\ w\rangle}$$

if – $(\mathit{\Gamma},\ t_1 < t_2\,, \mathit{\Delta})[m] =\ t_1 < t_2$
- for $i \in \{1,2\}$, $p_i \in \mathrm{Pos}(t_i)\backslash\{\varepsilon\}$ and $t_i/p_i = u_i$
- there are $\hat{t}_1, \hat{t}_2 \in \mathcal{T}(sig^C, V^{\mathrm{C}})$ and $x_1, x_2 \in V^{\mathrm{C}}\backslash\mathrm{Var}(t_1, t_2)$ 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 $x \in V^{\mathrm{C}}\backslash\{x_1, x_2\}$, $|\hat{t}_1|_x \leq |\hat{t}_2|_x$.

<-Transitivity:

`<-trans` $m\ w_2$

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

if $(\mathit{\Gamma},\ w_1 < w_3\,, \mathit{\Delta})[m] =\ w_1 < w_3$.

C.1.6 Non-Applicative Case Analyses

Substitution Addition:

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

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

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

Literal Addition:

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

$$\frac{\langle \mathit{\Gamma}\ ;\ w\rangle}{\langle \mathit{\Lambda}_1, \mathit{\Gamma}\ ;\ w\rangle\ \ \ldots\ \ \langle \mathit{\Lambda}_n, \mathit{\Gamma}\ ;\ w\rangle\ \langle \mathit{\Lambda}, \mathit{\Gamma}\ ;\ w\rangle}$$

if $\mathit{\Lambda}_1, \ldots, \mathit{\Lambda}_n, \mathit{\Lambda}$ is the case analysis resulting from literals $\lambda_1, \ldots, \lambda_n$ for $n > 0$.

C.2 Applicative Inference Rules

C.2.1 Non-Inductive Applicative Inference Rules

Non-Inductive Subsumption:[1]

`axiom-subs` *DefRule* μ

$$\frac{\langle \Gamma \,;\, w \rangle}{\langle \Lambda_1, \Gamma \,;\, w \rangle \;\ldots\; \langle \Lambda_n, \Gamma \,;\, w \rangle} \quad \text{with } \langle \Pi \,;\, () \rangle^{\mathcal{L}}$$

if there is a substitution μ and a 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.

`lemma-subs` *PS-Tree* μ

$$\frac{\langle \Gamma \,;\, w \rangle}{\langle \Lambda_1, \Gamma \,;\, w \rangle \;\ldots\; \langle \Lambda_n, \Gamma \,;\, w \rangle} \quad \text{with } \langle \Pi \,;\, \hat{w} \rangle^{\mathcal{L}}$$

if there is a substitution μ and a 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.

Non-Inductive Rewriting:

`axiom-rewrite` *m p DefRule n* μ

$$\frac{\langle \Gamma, \lambda, \Delta \,;\, w \rangle}{\langle \Lambda_1, \Gamma, \lambda, \Delta \,;\, w \rangle \;\ldots\; \langle \Lambda_n, \Gamma, \lambda, \Delta \,;\, w \rangle \; \langle \Lambda, \Gamma, \lambda[r\mu]_p, \Delta \,;\, w \rangle} \quad \text{with } \langle \Pi,\, l \doteq r \,, \Sigma \,;\, () \rangle^{\mathcal{L}}$$

if there is a position $p \in \mathrm{Pos}(\lambda)$, a substitution μ and a clause Θ such that

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

[1] QuodLibet provides two variants of Non-Inductive Subsumption and Rewriting, resp. — one for applying defining rules (`axioms`) and one for applying goals in the roots of proof state trees (`lemmas`).

`lemma-rewrite` $m\ p\ \mathit{PS\text{-}Tree}\ n\ \mu$

$$\frac{\langle \Gamma, \lambda, \Delta\,;\, w\rangle}{\langle \Lambda_1, \Gamma, \lambda, \Delta\,;\, w\rangle\ \ldots\ \langle \Lambda_n, \Gamma, \lambda, \Delta\,;\, w\rangle\ \langle \Lambda, \Gamma, \lambda[r\mu]_p, \Delta\,;\, w\rangle} \quad \text{with}\ \ \langle \Pi,\ l \doteq r\ , \Sigma\,;\, \hat{w}\rangle^{\mathcal{L}}$$

if there is a position $p \in \mathrm{Pos}(\lambda)$, a substitution μ and a clause Θ such that

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

Applicative Literal Removal:

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

$$\frac{\langle \Gamma, \lambda, \Delta\,;\, w\rangle}{\langle \Gamma, \Delta\,;\, w\rangle} \quad \text{with}\ \ \langle \Pi, \lambda', \Sigma\,;\, \hat{w}\rangle^{\mathcal{L}}$$

if there is a substitution μ such that

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

C.2.2 Inductive Applicative Inference Rules

Inductive Subsumption:

`ind-subs` $\mathit{PS\text{-}Tree}\ \mu$

$$\frac{\langle \Gamma\,;\, w\rangle}{\langle \Lambda_1, \Gamma\,;\, w\rangle\ \ldots\ \langle \Lambda_n, \Gamma\,;\, w\rangle\ \langle\, \hat{w}\mu < w\ , \Lambda, \Gamma\,;\, w\rangle} \quad \text{with}\ \ \langle \Pi\,;\, \hat{w}\rangle^{\mathcal{I}}$$

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

- the goal labeling the root of the proof state tree $\mathit{PS\text{-}Tree}$ 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.

Inductive Rewriting:

`ind-rewrite` $m\ p\ \mathit{PS\text{-}Tree}\ n\ \mu$

$$\frac{\langle \Gamma, \lambda, \Delta\, ;\, w\rangle}{\langle \Lambda_1, \Gamma, \lambda, \Delta\, ;\, w\rangle\ \ldots\ \langle \Lambda_n, \Gamma, \lambda, \Delta\, ;\, w\rangle\ \langle \Lambda, \Gamma, \lambda[r\mu]_p, \Delta\, ;\, w\rangle\ \langle\, \hat{w}\mu < w\,, \Lambda', \Gamma, \lambda, \Delta\, ;\, w\rangle}$$

with $\langle \Pi,\ l \doteq r\,, \Sigma\, ;\, \hat{w}\rangle^{\mathcal{I}}$

if there is a position $p \in \mathrm{Pos}(\lambda)$, a substitution μ and a clause Θ such that

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

Appendix D

Syntax of QML

This appendix contains a complete definition of the syntax of the proof control language QML in Extended Backus-Naur Form (refer to Appendix B for an explanation of EBNF productions). The start symbol for the syntax of QML is *QML*. Observe that the QML compiler of QuodLibet accepts both small and capital letters in terminal symbols.

QML Module

QML ::= "`MODULE`" *Module-Ident* "`;`"
  [*Export*]
  [*Import*]
  { *TypeDefinition* |
  VarDeclaration |
  TacticDefinition |
  ProcedureDefinition |
  FunctionDefinition }*
 "`END`" *Module-Ident* "`.`" .

Module Interface

Export ::= "`EXPORT`"
  ["`TYPE`" *Type-Ident* { "`,`" *Type-Ident* }* "`;`"]
  ["`VAR`" *Var-Ident* { "`,`" *Var-Ident* }* "`;`"]
  ["`ROUTINE`" *Routine-Ident* { "`,`" *Routine-Ident* }* "`;`"]
  ["`PUBLIC`" *Routine-Ident* { "`,`" *Routine-Ident* }* "`;`"]
 "`END`" "`EXPORT`" "`;`" .

Import ::= "`IMPORT`"
["`TYPE`" { *Module-Ident* ":"
Type-Ident { "," *Type-Ident* }* ";" }*]
["`VAR`" { *Module-Ident* ":"
Var-Ident { "," *Var-Ident* }* ";" }*]
["`ROUTINE`" { *Module-Ident* ":"
Routine-Ident { "," *Routine-Ident* }* ";" }*]
"`END`" "`IMPORT`" ";" .

Type Definition

TypeDefinition ::= "`TYPE`" { *Type-Ident* "`=`" *DefinitionType* ";" }+ .

DefinitionType ::= *ListType* | *StructType* | *EnumType* .

ListType ::= "`[`" *FormalType* "`]`" .

FormalType ::= *Type-Ident* | "`[`" *FormalType* "`]`" .

StructType ::= *ConstructorDef* { "`|`" *ConstructorDef* }* .

ConstructorDef ::= *Ident* "`(`" [*CtrParamSection* { ";" *CtrParamSection* }*] "`)`" .

CtrParamSection ::= *Component-Ident* { "," *Component-Ident* }* ":" *FormalType* .

EnumType ::= *Const-Ident* { "`|`" *Const-Ident* }* .

Variable Declaration

VarDeclaration ::= "`VAR`" { *Var-Ident* { "," *Var-Ident* }* ":" *FormalType* ";" }+ .

Routine Definition

TacticDefinition ::= "`TACTIC`" *Routine-Ident FormalParams* ";"
[*Description*]
[*AutoCall*]
[*VarDeclaration*]
"`BEGIN`"
StatementSequence
"`END`" *Routine-Ident* ";" .

ProcedureDefinition ::= “PROCEDURE” *Routine-Ident FormalParams* “;”
[*Description*]
[*VarDeclaration*]
“BEGIN”
StatementSequence
“END” *Routine-Ident* “;” .

FunctionDefinition ::= “FUNCTION” *Routine-Ident FormalParams* “:” *FormalType* “;”
[*Description*]
[*VarDeclaration*]
“BEGIN”
StatementSequence
“END” *Routine-Ident* “;” .

FormalParams ::= “(” [*ParamSection* { “;” *ParamSection* }*] “)” .

ParamSection ::= [“REF”] *Var-Ident* { “,” *Var-Ident* }* “:” *FormalType* .

Description ::= “DESCRIPTION” *String* .

AutoCall ::= “AUTOCALL” *Routine-Ident* { “,” *Routine-Ident* }* “;” .

Statement

StatementSequence ::= *Statement* { “;” *Statement* }* .

Statement ::= [*AssignmentStatement* |
ConditionalStatement |
ForStatement |
ForEachStatement |
WhileStatement |
RepeatStatement |
LoopStatement |
ExitStatement |
CallStatement |
ReturnStatement |
TryStatement |
FailStatement] .

AssignmentStatement ::= *Var-Ident* { “.” *Component-Ident* }* “:=” *Expression* .

$ConditionalStatement ::=$ "IF" $Expression$ "THEN" $StatementSequence$
  { "ELSIF" $Expression$ "THEN" $StatementSequence$ }*
  ["ELSE" $StatementSequence$]
"END" "IF" .

$ForStatement ::=$ "FOR" $Var\text{-}Ident$ ":=" $Expression$ "TO" $Expression$
  ["BY" $Expression$] "DO"
  $StatementSequence$
"END" "FOR" .

$ForEachStatement ::=$ "FOREACH" $Var\text{-}Ident$ "IN" $Expression$ "DO"
  $StatementSequence$
"END" "FOR" .

$WhileStatement ::=$ "WHILE" $Expression$ "DO"
  $StatementSequence$
"END" "WHILE" .

$RepeatStatement ::=$ "REPEAT"
  $StatementSequence$
"UNTIL" $Expression$.

$LoopStatement ::=$ "LOOP"
  $StatementSequence$
"END" "LOOP" .

$ExitStatement ::=$ "EXIT" .

$CallStatement ::=$ $Routine\text{-}Ident$ "(" [$Expression$ { "," $Expression$ }*] ")" .

$ReturnStatement ::=$ "RETURN" [$Expression$] .

$TryStatement ::=$ "TRY" $StatementSequence$
  ["FAILURE" $StatementSequence$]
  ["SUCCESS" $StatementSequence$]
"END" "TRY" .

$FailStatement ::=$ "FAIL" .

Expression

$Expression ::=$ $Factor$ { $Operator$ $Factor$ }* .

Factor ::= “`FALSE`” | “`TRUE`” |
Nat |
PositiveReal |
String |
Ident { “`.`” *Component-Ident* }* |
Routine-Ident “`(`” [*Expression* { “`,`” *Expression* }*] “`)`” |
(“`NOT`” | “`+`” | “`-`”) *Factor* |
“`[`” [*Expression* { “`,`” *Expression* }* [“`|`” *Expression*]] “`]`” |
“`(`” *Expression* “`)`” |
ProverObject .

Operator ::= “`OR`” |
“`AND`” |
“`=`” | “`/=`” | “`<`” | “`>`” | “`<=`” | “`>=`” |
“`AT`” | “`IN`” |
“`+`” | “`-`” | “`++`” |
“`*`” | “`/`” | “`DIV`” | “`MOD`” |
“`MATCH`” .

Prover Object

ProverObject ::= “`'`” (*Term* |
Weight |
Literal |
Formula |
Substitution |
Position) “`'`” .

Term ::= *Ident* [“`(`” *Term* { “`,`” *Term* }* “`)`”] |
Nat |
(“`$`” | “`%`”) *Var-Ident* .

Weight ::= “`(`” [*Term* { “`,`” *Term* }*] “`)`” |
(“`$`” | “`%`”) *Var-Ident* .

Literal ::= *Term* “`=`” *Term* |
Term “`=/=`” *Term* |
“`DEF`” *Term* |
“`~`” “`DEF`” *Term* |
(*Weight* | *Term*) “`<`” (*Weight* | *Term*) |
“`~`” “`(`” (*Weight* | *Term*) “`<`” (*Weight* | *Term*) “`)`” |
(“`$`” | “`%`”) *Var-Ident* .

Formula ::= "{" [*Literal* { "," *Literal* }*] "}" .

Substitution ::= "[" [["$"] *Ident* "<--" *Term*
{ "," ["$"] *Ident* "<--" *Term* }*] "]" .

Position ::= "ROOT" |
"[" *Nat* ["^" *Nat*] { ":" *Nat* ["^" *Nat*] }* "]" .

Identifier

Module-Ident ::= *Ident* .

Type-Ident ::= *Ident* .

Var-Ident ::= *Ident* .

Routine-Ident ::= *Ident* .

Component-Ident ::= *Ident* .

Const-Ident ::= *Ident* .

Token

Ident ::= *Letter* { *Letter* | *Digit* | *SpecialCharacter* }* .

String ::= """ { *StringCharacter* | "" "" }* """ .

Nat ::= { *Digit* }+ .

PositiveReal ::= { *Digit* }+ "." { *Digit* }+ [*ScaleFactor*] |
{ *Digit* }+ *ScaleFactor* .

ScaleFactor ::= "E" ["+" | "-"] { *Digit* }+ .

Letter ::= "a" | ... | "z" | "A" | ... | "Z" .

Digit ::= "0" | ... | "9" .

SpecialCharacter ::= "-" | "_" .

StringCharacter ::= All symbols with ASCII-code greater than 31
except for the symbol """ .

Appendix E

Examples of Proof Constructions

This appendix contains (the listings of) a collection of command files for various interesting proof constructions that can be executed by the command interpreter of QuodLibet with the `execute` command (see Section 7.3.3). Every command used in the following is explained either in Section 7.3 or in Section 8.2.2. For a description of the (QML) proof control routines occurring in these files — they are all preceded by "`call`" — refer to Section 8.3. Note that the characters "`//`" indicate comments in the QuodLibet command language. Moreover, the comment "`//qed`" means that the command in the preceding line is the last one needed for completing the proof for the given conjecture.

E.1 Truth Values and Natural Numbers

E.1.1 The Basic Specification

```
// basic-spec.ql

initialize
call initialize-database


// Sort and constructors for the truth values:

define sort Bool with constructors
   true  :     --> Bool
   false :     --> Bool.

declare constructor variables b : Bool.
declare general variables B : Bool.
```

```
// Properties of truth values:

prove { b = false, b = true } Bool-complete-1
call standard-strategy
//qed

call activate-lemma  Bool-complete-1


prove { ~def B, B = true, B = false } Bool-Complete-2
call standard-strategy
//qed


// Not-operator on truth values:

define operator not : Bool --> Bool
with defining rules
not-1:
  not(false) = true
not-2:
  not(true)  = false.

call analyze-operator not
// analyze-operator generates
// prove { def not(b) } not-def-auto
call standard-strategy
//qed

call activate-lemma  not-def-auto


// not(not(b)) = b:

prove { not(not(b)) = b } not-not-b
call standard-strategy
//qed

call activate-lemma not-not-b



// Sort and constructors for the natural numbers

define sort Nat with constructors
   0  : --> Nat
   s  : Nat --> Nat.
```

```
declare constructor variables x, x1, x2, y, y1, y2, z, z1, z2: Nat.
declare general variables X, Y, Z: Nat.


// the second of Peano's Postulates

prove { x = y, s(x) =/= s(y) } Nat-peano-2
call standard-strategy
//qed


// Less on the natural numbers:

define operator less : Nat Nat --> Bool
with defining rules
less-1:
  less(0,s(y))    = true
less-2:
  less(x,0)       = false
less-3:
  less(s(x),s(y)) = less(x,y).

call analyze-operator less
// analyze-operator generates
// prove { def less(x,y) } less-def-auto
call standard-strategy

call activate-lemma less-def-auto


// less-or-equal on the natural numbers:

define operator leq : Nat Nat --> Bool
with defining rules
leq-1:
  leq(0,y)       = true
leq-2:
  leq(s(x),0)    = false
leq-3:
  leq(s(x),s(y)) = leq(x,y).

call analyze-operator leq
// analyze-operator generates
// prove { def leq(x,y) } leq-def-auto
call standard-strategy
//qed

call activate-lemma leq-def-auto
```

```
// Properties of the less- and leq- predicates

prove { less(x,s(x)) = true } less-x-sx
call standard-strategy
//qed

call activate-lemma  less-x-sx


prove { less(x,x) = false } less-x-x
call standard-strategy
//qed

call activate-lemma less-x-x


prove { leq(x,x) = true } leq-x-x

call standard-strategy
//qed

call activate-lemma leq-x-x


prove { leq(x,y) = true, leq(y,x) = true } leq-complete
call standard-strategy
//qed

call activate-lemma leq-complete


prove { less(x,y) = true, leq(y,x) = true } less-or-leq
call standard-strategy
//qed

call activate-lemma less-or-leq


prove { less(0,y) = true, y = 0 } less-0-y
call standard-strategy
//qed

call activate-lemma less-0-y


prove { less(x,z) = true, less(x,y) =/= true, less(y,z) =/= true }
      less-transitive
call standard-strategy
```

```
//qed

call activate-lemma less-transitive


prove { leq(x,z) = true, leq(x,y) =/= true, leq(y,z) =/= true }
      leq-transitive
call standard-strategy
//qed

call activate-lemma leq-transitive


prove { x = y, less(x,y) = true, less(y,x) = true } less-trichotomy
call restricted-strategy
apply ind-rewrite 1 [1:1] less-trichotomy 1 []
call simplify-goal
call set-weights
call prove-order-subgoal
//qed

call activate-lemma less-trichotomy


prove { x < y, less(x,y) =/= true } ind-less
call restricted-strategy
apply <-mono 1 [1] [1]
call apply-ind-hypothesis
call set-weights
call prove-order-subgoal
//qed

call activate-lemma ind-less
```

E.1.2 Addition and Multiplication

```
// plus-times.ql

// requires: basic-spec.ql


// Addition on the natural numbers:

define operator plus: Nat Nat --> Nat
with defining rules
plus-1:
  plus(x,0)    = x
```

```
plus-2:
  plus(x,s(y)) = s(plus(x,y)).

call analyze-operator plus
// analyze-operator generates
// prove { def plus(x,y) } plus-def-auto
call standard-strategy
//qed

call activate-lemma plus-def-auto


// Properties of plus:

prove { plus(0,y) = y } plus-0-y
call standard-strategy
//qed

call activate-lemma plus-0-y


prove { plus(s(x),y) = s(plus(x,y)) } plus-sx-y
call standard-strategy
//qed

call activate-lemma plus-sx-y


prove { plus(plus(x,y),z) = plus(x,plus(y,z)) } plus-assoc
call standard-strategy
//qed

call activate-lemma plus-assoc


prove { plus(x,y) = plus(y,x) } plus-com
call standard-strategy
//qed


prove { less(x,plus(x,y)) = true, y = 0 } less-x-plus-x-y

call restricted-strategy
apply lemma-subs less-transitive [ y <-- plus(x,y), z <-- s(plus(x,y)) ]
call prove-def-subgoal
call prove-def-subgoal
apply ind-subs less-x-plus-x-y []
call simplify-goal
```

```
set current g-node [1:2:1^5:4]
call simplify-goal
call set-weights
call prove-order-subgoal
//qed

call activate-lemma less-x-plus-x-y


prove { less(x,plus(y,z)) = true, less(x,y) =/= true } less-x-plus-y-z

call expand-operator 2 [1]
call simplify-goal
call simplify-goal
call apply-ind-hypothesis
call set-weights
call prove-order-subgoal
//qed

// Note that standard-strategy expands  plus(y,z)  instead of  less(x,y)
// and the proof construction diverges.

call activate-lemma less-x-plus-y-z


prove { less(plus(x,z),plus(y,z)) = true, less(x,y) =/= true } plus-mono
call standard-strategy
//qed


prove { plus(x,z) =/= plus(y,z), x = y } plus-cancel

call restricted-strategy
apply lemma-subs Nat-peano-2 [ x <-- plus(x,z), y <-- plus(y,z) ]
call prove-def-subgoal
call prove-def-subgoal
call apply-ind-hypothesis
call set-weights
call prove-order-subgoal
//qed

call activate-lemma plus-cancel
```

```
// Multiplication on the natural numbers:

define operator times: Nat Nat --> Nat
with defining rules
times-1:
  times(x,0)    = 0
times-2:
  times(x,s(y)) = plus(times(x,y),x).

call analyze-operator times
// analyze-operator generates
// prove { def times(x,y) } plus-def-auto
call standard-strategy
//qed

call activate-lemma times-def-auto


// Properties of times:

prove { times(0,y) = 0 } times-0-y
call standard-strategy
//qed

call activate-lemma times-0-y


prove { times(s(x),y) = plus(times(x,y),y) } times-sx-y

call restricted-strategy
apply lemma-rewrite 4 [2:1:2] plus-com 1 []
call simplify-goal
//qed

call activate-lemma times-sx-y


prove { times(x,plus(y,z)) = plus(times(x,y),times(x,z)) } times-plus-dist
call standard-strategy
//qed

call activate-lemma times-plus-dist


prove { times(times(x,y),z) = times(x,times(y,z)) } times-assoc
call standard-strategy
//qed
```

```
call activate-lemma times-assoc


prove { times(x,y) = times(y,x) } times-com
call standard-strategy
//qed
```

E.1.3 (Partial) Subtraction and Division

```
// minus-div.ql

// requires: basic-spec.ql


// (Partial) subtraction on the natural numbers:

define operator minus: Nat Nat --> Nat
with defining rules
minus-1:
  minus(x,0)       = x
minus-2:
  minus(s(x),s(y)) = minus(x,y).

call analyze-operator minus
// analyze-operator does NOT suggest a domain lemma for minus.
// Domain lemma for minus:

prove { def minus(x,y), less(x,y) = true } minus-def
call standard-strategy
//qed
call activate-lemma minus-def


// Induction lemma for minus:

prove { minus(x,y) < x, less(x,y) = true, y = 0 } minus-ind-lma

call restricted-strategy
apply <-trans 1 x
apply ind-subs minus-ind-lma []
call simplify-goal
set current g-node minus-ind-lma [1:3:1^7:2]
call simplify-goal
call set-weights
call prove-order-subgoal
//qed
```

```
call activate-lemma minus-ind-lma

// x - x = 0 :

prove { minus(x,x) = 0 } minus-x-x
call standard-strategy
//qed

call activate-lemma minus-x-x


// (Partial) division on the natural numbers:

define operator div: Nat Nat --> Nat
with defining rules
div-1:
  div(x,y) = 0
  if
    y =/= 0,
    less(x,y) = true
div-2:
  div(x,y) = s(div(minus(x,y),y))
  if
    y =/= 0,
    less(x,y) =/= true,
    def less(x,y).

call analyze-operator div
// analyze-operator generates
// { def div(x,y), y = 0 } div-def-auto
call standard-strategy
//qed

call activate-lemma div-def-auto


// x div 1 = x :

prove { div(x,s(0)) = x } div-x-1
call standard-strategy
//qed

call activate-lemma div-x-1
```

```
// x =/= 0 --> x div x = 1 :

prove { div(x,x) = s(0), x = 0 } div-x-x-1

call restricted-strategy
apply axiom-rewrite 1 [1:1] div-1 1 [x <-- 0, y <-- x]
call simplify-goal
call simplify-goal
//qed

call activate-lemma div-x-x-1
```

E.1.4 Ackermann's Function

```
// ack.ql

// requires:  basic-spec.ql

// Ackermann's function:

define operator ack: Nat Nat --> Nat
with defining rules
ack-1:
  ack(0,y)       = s(y)
ack-2:
  ack(s(x),0)    = ack(x,s(0))
ack-3:
  ack(s(x),s(y)) = ack(x,ack(s(x),y)).


call analyze-operator ack
// op-analyze generates
// prove { def ack(x,y) } ack-def-auto
call standard-strategy
//qed

call activate-lemma ack-def-auto


// Properties of ack:

prove { less(0,ack(x,y)) = true } ack-positive
call standard-strategy
//qed

// Note that the strategy finds a proof by structural induction on x
// (instead of (x,y))
```

```
call activate-lemma ack-positive


prove { less(y,ack(x,y)) = true }  less-y-ack-x-y

call restricted-strategy

assume { less(s(x),z) = true,
         less(x,y) =/= true,
         less(y,z) =/= true }
       less-trans-1
apply lemma-subs less-trans-1 [ x <-- y,
                                y <-- ack(s(x),y),
                                z <-- ack(x,ack(s(x),y)) ]
call prove-def-subgoal
call prove-def-subgoal
call apply-ind-hypothesis
set current g-node [1:3:1^3:4]
call apply-ind-hypothesis
set weight (x,y)
call prove-order-subgoal
call prove-order-subgoal
//qed (relative to less-trans-1)

call standard-strategy less-trans-1
//qed
```

E.1.5 A Non-Terminating Operation

```
// div1.ql

// requires: basic-spec.ql, plus-times.ql, minus-div.ql


// A non-terminating tail-recursive variant of the division:

define operator div1: Nat Nat Nat Nat --> Nat
with defining rules
div1-1:
  div1(x,y,z1,z2) = z1
  if
    x = z2
div1-2:
  div1(x,y,z1,z2) = div1(x,y,s(z1),plus(z2,y))
  if
    x =/= z2.
```

```
call analyze-operator div1
// analyze-operator does not conjecture that div1 is terminating.
// Hence, no domain lemma for div1 is suggested.
// A conjecture about div1:

prove { y = 0, div1(times(y,z),y,0,0) = z } div1-THM


// An adequate computational "invariant" for div1-THM:

assume { y = 0,
         less(x,times(y,z)) = true,
         div1(x,y,0,0) = div1(x,y,z,times(y,z)) }
       div1-invariant


// The proof of div1-THM (relative to div1-invariant):

apply lemma-rewrite 2 [1] div1-invariant 3 [x <-- times(y,z)]
call prove-def-subgoal
call simplify-goal
call simplify-goal
//qed (relative to div1-invariant)


// The proof of div1-invariant:

set current ps-tree div1-invariant

call restricted-strategy
apply ind-rewrite 3 [1] div1-invariant 3 []
call simplify-goal
apply axiom-rewrite 4 [1] div1-2 1 [z1 <-- z, z2 <-- times(y,z)]
call prove-def-subgoal
apply const-rewrite 5 1 [1:1]
call simplify-goal
call simplify-goal
call set-weights
call prove-order-subgoal
//qed
```

E.2 Lists of Natural Numbers

E.2.1 Basic Definitions

```
// nat-list.ql

// requires: basic-spec.ql

// Sort and constructors for lists of natural numbers:

define sort ListNat with constructors
   nil  :              --> ListNat
   cons : Nat ListNat --> ListNat.

declare constructor variables l, l1, l2, l3: ListNat.


// Measure-function for lists:

define operator length : ListNat --> Nat
with defining rules
  length-1  : length(nil)       = 0
  length-2  : length(cons(x,l)) = s(length(l)).

call analyze-operator length
// analyze-operator generates
// prove { def length(l) } length-def-auto
call standard-strategy
//qed

call activate-lemma  length-def-auto
```

E.2.2 Some Operations on Lists

```
// list-operators.ql

// requires: basic-spec.ql, nat-list.ql


// Append on lists of natural numbers:

define operator append: ListNat ListNat --> ListNat
with defining rules
  app-1:  append(nil,l2)        = l2
  app-2:  append(cons(x,l1),l2) = cons(x,append(l1,l2)).
```

```
call analyze-operator append
// analyze-operator generates
// prove { def append(l1,l2) } append-def-auto
call standard-strategy
//qed

call activate-lemma append-def-auto


// Properties of append:

prove { append(l,nil) = l } append-l-nil
call standard-strategy
//qed

call activate-lemma append-l-nil


prove { append(append(l1,l2),l3) = append(l1,append(l2,l3)) } append-assoc
call standard-strategy
//qed

call activate-lemma append-assoc


// Reverse on lists of natural numbers:

define operator reverse : ListNat --> ListNat
with defining rules
reverse-1:
  reverse(nil) = nil
reverse-2:
  reverse(cons(x,l)) = append(reverse(l),cons(x,nil)).

call analyze-operator reverse
// analyze-operator generates
// prove { def reverse(l) } reverse-def-auto
call standard-strategy
//qed

call activate-lemma reverse-def-auto


// Properties of append and reverse:

prove { reverse(append(l,cons(x,nil))) = cons(x,reverse(l)) } lma-rev-app
call standard-strategy
//qed
```

```
call activate-lemma lma-rev-app


prove { reverse(reverse(l)) = l } rev-rev-l
call standard-strategy
//qed

call activate-lemma rev-rev-l
// List inclusion:

define operator elem : Nat ListNat --> Bool
with defining rules
elem-1:
  elem(x,nil) = false
elem-2:
  elem(x,cons(y,l)) = true
  if
    x = y
elem-3:
  elem(x,cons(y,l)) = elem(x,l)
  if
    x =/= y.

call analyze-operator elem
// analyze-operator generates
// prove { def elem(x,l) } elem-def-auto
call standard-strategy
//qed

call activate-lemma elem-def-auto

// Properties of elem and append:

prove { elem(x,append(l1,l2)) = true, elem(x,l1) =/= true } elem-app-1
call standard-strategy
//qed


prove { elem(x,append(l1,l2)) = true, elem(x,l2) =/= true } elem-app-2

call restricted-strategy
apply lit-add x = x1.
call simplify-goal
call simplify-goal
call apply-ind-hypothesis
call set-weights
call prove-order-subgoal
//qed
```

```
prove { elem(x,append(l1,l2)) =/= true,
        elem(x,l1) = true,
        elem(x,l2) = true } elem-app-3
call standard-strategy
//qed
// The nth-elem function (returns the n-th element of a list if it exists)

define operator nth-elem : Nat ListNat --> Nat
with defining rules
nth-1:
  nth-elem(s(0),cons(y,l)) = y
nth-2:
  nth-elem(s(s(x)),cons(y,l)) = nth-elem(s(x),l).

call analyze-operator nth-elem
// analyze-operator does NOT suggest a domain lemma for nth-elem.


// Domain lemma for nth-elem:

prove { def nth-elem(x,l), x = 0, less(length(l),x) = true } nth-elem-def
call standard-strategy
//qed


// A property of elem and nth-elem :

prove { elem(y,l) = true,
        x = 0,
        less(length(l),x) = true,
        nth-elem(x,l) =/= y }
      elem-nth-elem
call standard-strategy
//qed
```

E.2.3 Mergesort

```
//mergesort.ql

//requires:  basic-spec.ql, nat-list.ql


// Operators for the mergesort algorithm:

// merge:

define operator merge : ListNat ListNat --> ListNat
with defining rules
merge-1:
  merge(nil,l) = l
merge-2:
  merge(l,nil) = l
merge-3:
  merge(cons(x,l1),cons(y,l2)) = cons(x,merge(l1,cons(y,l2)))
  if
    leq(x,y) = true
merge-4:
  merge(cons(x,l1),cons(y,l2)) = cons(y,merge(cons(x,l1),l2))
  if
    leq(x,y) =/= true,
    def leq(x,y).

call analyze-operator merge
// analyze-operator generates
// prove { def merge(l1,l2) } merge-def-auto
call standard-strategy
//qed

call activate-lemma  merge-def-auto


// split1:

define operator split1 : ListNat --> ListNat
with defining rules
split1-1:
  split1(nil) = nil
split1-2:
  split1(cons(x,nil)) = cons(x,nil)
split1-3:
  split1(cons(x,cons(y,l))) = cons(x,split1(l)).
```

```
call analyze-operator split1
// analyze-operator generates
// prove { def split1(l) } split1-def-auto
call standard-strategy
//qed

call activate-lemma  split1-def-auto


// Two lemmas for the proof of the induction lemma for split1

prove { s(s(x)) =/= s(0) } s-s-x-s-0
call standard-strategy
//qed

call activate-lemma s-s-x-s-0


prove { split1(l) = l, length(l) =/= s(0) } split1-len-1
call standard-strategy
//qed

call activate-lemma split1-len-1


// Induction lemma for the destructor split1:

prove { split1(l) < l, l = nil, length(l) = s(0) } split1-ind-lma

call restricted-strategy
apply <-mono 1 [2] [2:2]
apply ind-subs split1-ind-lma []
call simplify-goal
call simplify-goal
call set-weights
call prove-order-subgoal
//qed

call activate-lemma split1-ind-lma

// split2:

define operator split2 : ListNat --> ListNat
with defining rules
split2-1:
  split2(nil) = nil
split2-2:
  split2(cons(x,nil)) = nil
```

```
split2-3:
  split2(cons(x,cons(y,l))) = cons(y,split2(l)).

call analyze-operator split2
// analyze-operator generates
// prove { def split2(l) } split2-def-auto
call standard-strategy
//qed

call activate-lemma  split2-def-auto


// Induction lemma for the destructor split2:

prove { split2(l) < l, l = nil } split2-ind-lma

call restricted-strategy
apply <-mono 1 [2] [2:2]
apply ind-subs split2-ind-lma []
call simplify-goal
call set-weights
call prove-order-subgoal
//qed

call activate-lemma split2-ind-lma


// mergesort:

define operator mergesort : ListNat --> ListNat
with defining rules
mergesort-1:
  mergesort(nil) = nil
mergesort-2:
  mergesort(cons(x,nil)) = cons(x,nil)
mergesort-3:
  mergesort(cons(x,cons(y,l))) =
      merge(mergesort(split1(cons(x,cons(y,l)))),
            mergesort(split2(cons(x,cons(y,l))))).

call analyze-operator mergesort
// analyze-operator generates
// prove { def mergesort(l) } mergesort-def-auto
call standard-strategy
//qed

call activate-lemma  mergesort-def-auto
```

```
// elemleqlist-p(x,l)  iff  leq(x,y) for each y in l:

define operator elemleqlist-p : Nat ListNat --> Bool
with defining rules
eleql-p-1:
  elemleqlist-p(x,nil) = true
eleql-p-2:
  elemleqlist-p(x,cons(y,l)) = elemleqlist-p(x,l)
  if
    leq(x,y) = true
eleql-p-3:
  elemleqlist-p(x,cons(y,l)) = false
  if
    leq(x,y) =/= true,
    def leq(x,y).

call analyze-operator elemleqlist-p
// analyze-operator generates
// prove { def elemleqlist-p(x,l) } elemleqlist-p-def-auto
call standard-strategy
//qed

call activate-lemma elemleqlist-p-def-auto


// A predicate for sorted lists:

define operator sorted-p : ListNat --> Bool
with defining rules
sorted-p-1:
  sorted-p(nil) = true
sorted-p-2:
  sorted-p(cons(x,l)) = sorted-p(l)
  if
    elemleqlist-p(x,l) = true
sorted-p-3:
  sorted-p(cons(x,l)) = false
  if
    elemleqlist-p(x,l) =/= true,
    def elemleqlist-p(x,l).

call analyze-operator sorted-p
// analyze-operator generates
// prove { def sorted-p(l) } sorted-p-def-auto
call standard-strategy
//qed

call activate-lemma sorted-p-def-auto
```

```
// A correctness property of mergesort:

prove { sorted-p(mergesort(l)) = true } mergesort-correct

call restricted-strategy
apply axiom-rewrite 1 [1] sorted-p-2 1 [l <-- nil]
call simplify-goal
call simplify-goal

// The key lemma in the proof:

assume { sorted-p(merge(l1,l2)) = true,
         sorted-p(l1) =/= true,
         sorted-p(l2) =/= true } sorted-merge
apply lemma-rewrite 1 [1] sorted-merge 1
      [l1 <-- mergesort(cons(x,split1(l))) ,
      l2 <-- mergesort(cons(y,split2(l)))]
call prove-def-subgoal
call prove-def-subgoal
call apply-ind-hypothesis
set current g-node [1:3:1^7:4]
call apply-ind-hypothesis
set current g-node [1:3:1^7:5]
call prove-tautology
set weight l
apply <-mono 1 [2] [2:2]
call prove-order-subgoal
apply <-mono 1 [2] [2:2]
call prove-order-subgoal
//qed (relative to sorted-merge)


// Two lemmas (needed for sorted-merge) and their proofs:

prove { elemleqlist-p(x,merge(l1,l2)) = true,
        elemleqlist-p(x,l1) =/= true,
        elemleqlist-p(x,l2) =/= true } elemleqlist-merge

call restricted-strategy
apply ind-subs elemleqlist-merge [ l2 <-- cons(x1,l2) ]
call simplify-goal
set current g-node elemleqlist-merge [1:5:1:2:1:2:1^9:2]
apply ind-subs elemleqlist-merge [ l1 <-- cons(y,l1) ]
call simplify-goal
set weight (l1,l2)
call prove-order-subgoal
call prove-order-subgoal
//qed
```

```
call activate-lemma elemleqlist-merge


prove { elemleqlist-p(x,l) = true,
        leq(x,y) =/= true,
        elemleqlist-p(y,l) =/= true } elemleqlist-trans
call standard-strategy
//qed

call activate-lemma elemleqlist-trans



// The proof of lemma sorted-merge:

set current ps-tree sorted-merge

call restricted-strategy
apply axiom-rewrite 4  [1] sorted-p-2 1 [l <-- merge(l1,cons(y,l2))]
call simplify-goal
apply lemma-subs elemleqlist-merge [l2 <-- cons(y,l2)]
call simplify-goal
apply ind-subs sorted-merge [ l2 <-- cons(y,l2) ]
call simplify-goal
set current g-node [1:5:1:2:1:2:1^8]
apply axiom-rewrite 5  [1] sorted-p-2 1 [x <-- y ,
                                         l <-- merge(cons(x,l1),l2)]
call simplify-goal
apply lemma-subs elemleqlist-merge [x <-- y, l1 <-- cons(x,l1)]
apply lemma-subs leq-complete []
call simplify-goal
apply lemma-subs elemleqlist-trans [x <-- y , y <-- x , l <-- l1]
call simplify-goal
apply ind-subs sorted-merge [ l1 <-- cons(x,l1) ]
call simplify-goal
set weight (l1,l2)
call prove-order-subgoal
call prove-order-subgoal
//qed
```

E.3 Binary Trees with Natural Numbers

E.3.1 Basic Definitions

```
// nat-btrees.spec

// requires: basic-spec.ql, plus-times.ql


// Sort and constructors for binary trees with natural numbers:

define sort BTreeNat with constructors
  empty-btree :                        --> BTreeNat
  btree       : BTreeNat Nat BTreeNat --> BTreeNat.

declare constructor variables t, t1, t2, t3, t4: BTreeNat.



// Measure-function for binary trees:

define operator btree-size : BTreeNat --> Nat
with defining rules
btree-size-1:
  btree-size(empty-btree)    = 0
btree-size-2:
  btree-size(btree(t1,x,t2)) = s(plus(btree-size(t1),btree-size(t2))).

call analyze-operator btree-size
// analyze-operator generates
// prove { def btree-size(t1) } btree-size-def
call standard-strategy
//qed

call activate-lemma  btree-size-def
```

E.3.2 Search Trees

```
// search-trees.spec

// requires: basic-spec.ql, plus-times.ql, nat-btrees.spec


// Binary search-trees (with natural numbers as keys)

// Operators for the smallest (greatest) element in a search-tree:
```

```
define operator leftmost-elem : BTreeNat --> Nat
with defining rules
l-elem-1:
  leftmost-elem(btree(empty-btree,x,t)) = x
l-elem-2:
  leftmost-elem(btree(btree(t1,x,t2),y,t)) =
               leftmost-elem(btree(t1,x,t2)).

call analyze-operator leftmost-elem
// analyze-operator generates
// prove { def leftmost-elem(t1), t1 = empty-btree } leftmost-elem-def
call standard-strategy
//qed

call activate-lemma leftmost-elem-def


define operator rightmost-elem : BTreeNat --> Nat
with defining rules
r-elem-1:
  rightmost-elem(btree(t,x,empty-btree)) = x
r-elem-2:
  rightmost-elem(btree(t,x,btree(t1,y,t2))) =
                rightmost-elem(btree(t1,y,t2)).

call analyze-operator rightmost-elem
// analyze-operator generates
// prove { def rightmost-elem(t), t = empty-btree } rightmost-elem-def
call standard-strategy
//qed

call activate-lemma rightmost-elem-def


// A predicate for search-trees:

define operator search-tree-p : BTreeNat --> Bool
with defining rules

s-tree-p-1:
  search-tree-p(empty-btree) = true
s-tree-p-2:
  search-tree-p(btree(t1,x,t2)) = true
  if
    t1 = empty-btree,
    t2 = empty-btree
```

```
s-tree-p-3:
  search-tree-p(btree(t1,x,t2)) = true
  if
    t1 =/= empty-btree,
    search-tree-p(t1) = true,
    less(rightmost-elem(t1),x) = true,
    t2 = empty-btree
s-tree-p-4:
  search-tree-p(btree(t1,x,t2)) = true
  if
    t1 = empty-btree,
    t2 =/= empty-btree,
    search-tree-p(t2) = true,
    less(x,leftmost-elem(t2)) = true
s-tree-p-5:
  search-tree-p(btree(t1,x,t2)) = true
  if
    t1 =/= empty-btree,
    search-tree-p(t1) = true,
    less(rightmost-elem(t1),x) = true,
    t2 =/= empty-btree,
    search-tree-p(t2) = true,
    less(x,leftmost-elem(t2)) = true
s-tree-p-6:
  search-tree-p(btree(t1,x,t2)) = false
  if
    t1 =/= empty-btree,
    search-tree-p(t1) =/= true,
    def search-tree-p(t1)
s-tree-p-7:
  search-tree-p(btree(t1,x,t2)) = false
  if
    t2 =/= empty-btree,
    search-tree-p(t2) =/= true,
    def search-tree-p(t2)
s-tree-p-8:
  search-tree-p(btree(t1,x,t2)) = false
  if
    t1 =/= empty-btree,
    less(rightmost-elem(t1),x) =/= true,
    def less(rightmost-elem(t1),x)
s-tree-p-9:
  search-tree-p(btree(t1,x,t2)) = false
  if
    t2 =/= empty-btree,
    less(x,leftmost-elem(t2)) =/= true,
    def less(x,leftmost-elem(t2)).
```

```
call analyze-operator search-tree-p
// analyze-operator generates
// prove { def search-tree-p(t1) } search-tree-p-def
call standard-strategy
//qed

call activate-lemma  search-tree-p-def
```

E.3.3 A Flatten Operation

```
// flatten.ql

// requires: basic-spec.ql, plus-times.ql, nat-btrees.ql


// A flatten operation on binary trees with natural numbers:

define operator flatten : BTreeNat --> BTreeNat
with defining rules
flat-1:
  flatten(empty-btree) = empty-btree
flat-2:
  flatten(btree(empty-btree,x,t)) =
         btree(empty-btree,x,flatten(t))
flat-3:
  flatten(btree(btree(t1,x1,t2),x,t)) =
         flatten(btree(t1,x1,btree(t2,x,t))).

call analyze-operator flatten
// analyze-operator does not conjecture that  flatten  is terminating.
// Hence, no domain lemma is suggested.


//  A measure function for the "termination proof" of flatten:

define operator m : BTreeNat --> Nat
with defining rules
m-1:
  m(empty-btree) = 0
m-2:
  m(btree(t1,x,t2)) = s(plus(m(t1),plus(m(t1),m(t2)))).
```

```
call analyze-operator m
// analyze-operator generates
// prove { def m(t1) } m-def
call standard-strategy
//qed

call activate-lemma  m-def


// Lemmas for the "termination proof" and their proofs:

prove { less(plus(x,y),s(plus(x,z))) = less(y,s(z)) }  less-plus-1

apply subst-add [ x <-- 0 ] [ x <-- s(x) ].
call simplify-goal
call simplify-goal
call apply-ind-hypothesis
call set-weights
call prove-order-subgoal
//qed

// Note that standard-strategy expands  plus(x,y)  and  plus(x,z),
// and the proof construction diverges.

call activate-lemma less-plus-1


prove { less(x,s(plus(y,plus(z,x)))) = true } less-plus-2
call standard-strategy
//qed

call activate-lemma less-plus-2


// Domain lemma for flatten
// (includes a "termination proof" of flatten):

prove { def flatten(t) } flatten-def

apply subst-add
    [ t <-- empty-btree ]
    [ t <-- btree(empty-btree,x,t) ]
    [ t <-- btree(btree(t1,x1,t2),x,t) ].

call simplify-goal
```

```
call simplify-goal
apply ind-subs flatten-def []

set current g-node [1:3]
call simplify-goal
apply ind-subs flatten-def [ t <-- btree(t1,x1,btree(t2,x,t)) ]

// The "termination proof":

set weight m(t)

set current g-node [1:2:1^6]
call simplify-goal

call simplify-goal
apply lemma-subs ind-less
  [ x <-- s(s(plus(m(t1),
                   plus(m(t1),plus(m(t2),plus(m(t2),m(t))))))),
    y <-- s(s(s(plus(m(t1),
                     plus(m(t1),
                          plus(m(t2),
                               plus(m(t1),
                                    plus(m(t1),
                                         plus(m(t2),m(t))))))))) ]
call prove-def-subgoal
call prove-def-subgoal
call simplify-goal
//qed
```

Bibliography

Aubin, R. (1976). Mechanizing structural induction. PhD Thesis, University of Edinburgh.

Avenhaus, J. (1995). *Reduktionssysteme*. Springer, Berlin Heidelberg. (German).

Avenhaus, J., & Madlener, K. (1990). Term rewriting and equational reasoning. In Banerji, R. (Ed.), *Formal Techniques in Artificial Intelligence*, pp. 1–43. North-Holland, Amsterdam.

Avenhaus, J., & Madlener, K. (1997). Theorem proving in hierarchical clausal specifications. In Du, D., & Ko, K. (Eds.), *Advances in Algorithms, Languages, and Complexity*, pp. 1–52. Kluwer Academic Press.

Bachmair, L. (1988). Proof by consistency in equational theories. In 3^{rd} *LICS*, pp. 228–233.

Bergstra, J. A., & Klop, J. W. (1986). Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Science*, *32*, 323–362.

Bibel, W., & Eder, E. (1993). Methods and calculi for deduction. In Gabbay, D. M., Hogger, C. J., & Robinson, J. A. (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 1, chap. 3, pp. 67–182. Oxford University Press, Oxford.

Bouhoula, A. (1996). Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, *170*, 245–276.

Bouhoula, A., & Rusinowitch, M. (1995). Implicit induction in conditional theories. *Journal of Automated Reasoning*, *14*, 189–235.

Boyer, R. S., & Moore, J. S. (1979). *A Computational Logic*. Academic Press, New York.

Boyer, R. S., & Moore, J. S. (1988). *A Computational Logic Handbook*. Academic Press, New York.

Boyer, R. S., & Moore, J. S. (1990). A theorem prover for a computational logic. In Stickel, M. E. (Ed.), *Proc. of the* 10^{th} *International Conference on Automated Deduction*, Vol. 449 of *LNCS*, pp. 1–15. Springer.

Bronsard, F., Reddy, U. S., & Hasker, R. W. (1994). Induction using term orderings. In Bundy, A. (Ed.), *Proc. of the* 12^{th} *International Conference on Automated Deduction*, Vol. 814 of *LNAI*, pp. 102–117. Springer.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., & Smaill, A. (1993). Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, *62*, 185–253.

Bundy, A., van Harmelen, F., Horn, C., & Smaill, A. (1990). The Oyster-Clam system. In Stickel, M. E. (Ed.), *Proc. of the* 10^{th} *International Conference on Automated Deduction*, Vol. 449 of *LNCS*, pp. 647–648. Springer.

Constable, R. L. (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ.

DeRemer, F., & Kron, H. (1976). Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, *2*, 80–86.

Dershowitz, N., Okada, M., & Sivakumar, G. (1988). Confluence of conditional rewrite systems. In 1^{st} *CTRS*, Vol. 308 of *LNCS*, pp. 31–44. Springer.

Dershowitz, N. (1987). Termination of rewriting. *Journal of Symbolic Computation*, *3*, 69–116.

Dershowitz, N., & Jouannaud, J.-P. (1990). Rewrite systems. In van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chap. 6, pp. 243–320. Elsevier Science Publishers B. V., Amsterdam.

Ehrig, H., & Mahr, B. (1985). *Fundamentals of Algebraic Specification*, Vol. 1. Springer, Berlin Heidelberg New York Tokyo.

Embacher, C. (1995). Entwurf und Implementierung eines prototypischen Induktionsbeweisers für positiv/negativ bedingte Rewrite-Spezifikationen — eine softwaretechnische Realisierung. Projektarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, Germany.

Garland, S. J., & Guttag, J. V. (1989). An overview of LP, the Larch Prover. In *Proc. of the* 3^{rd} *International Conference on Rewriting Techniques and Applications*, Vol. 355 of *LNCS*, pp. 137–151. Springer.

Garland, S. J., & Guttag, J. V. (1991). A guide to LP, the Larch Prover. Technical report 82, DEC-SRC.

Gerberding, S., & Noltemeier, J. A. (1997). Incremental proof planning by meta-rules. In *Proc.* 10^{th} *Florida Artificial Intelligence Research Symposium (FLAIRS-97)*, pp. 171–175 Daytona Beach. ISBN 0-9620-1739-6.

Gordon, M. J., Milner, R., & Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanized Logic of Computation*, Vol. 78 of *LNCS*. Springer.

Gordon, M. J. C., & Melham, T. F. (1993). *Introduction to HOL — A theorem proving environment for higher order logic.* Cambridge University Press, Cambridge.

Gramlich, B., & Lindner, W. (1991). A guide to UNICOM, an inductive theorem prover based on rewriting and completion techniques. Seki-report SR-91-17, Fachbereich Informatik, Universität Kaiserslautern, Germany.

Gramlich, B. (1990). Completion based inductive theorem proving: A case study in verifying sorting algorithms. Seki-report SR-90-04, Fachbereich Informatik, Universität Kaiserslautern, Germany.

Hua, X., & Zhang, H. (1992). FRI: Failure-resistant induction in RRL. In Kapur, D. (Ed.), *Proc. of the* 11^{th} *International Conference on Automated Deduction*, Vol. 607 of *LNAI*, pp. 691–694. Springer.

Hutter, D. (1990). Guiding induction proofs. In Stickel, M. E. (Ed.), *Proc. of the* 10^{th} *International Conference on Automated Deduction*, Vol. 449 of *LNCS*, pp. 147–161.

Hutter, D. (1997). Colouring terms to control equational reasoning. *Journal of Automated Reasoning*, *18*, 399–442.

Hutter, D., & Sengler, C. (1996). INKA: The next generation. In McRobbie, M., & Slaney, J. (Eds.), *Proc. of the* 13^{th} *International Conference on Automated Deduction*, Vol. 1104 of *LNAI*, pp. 288–292. Springer.

Ireland, A., & Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, *16*, 79–111.

Jalote, P. (1991). *An Integrated Approach to Software Engineering.* Springer, New York.

Kapur, D., Nie, X., & Musser, D. R. (1994). An overview of the Tecton proof system. *Theoretical Computer Science*, *133*, 307–339.

Kapur, D., & Subramaniam, M. (1996a). Automating induction over mutually recursive functions. In Wirsing, M., & Nivat, M. (Eds.), *Proc. of the* 5^{th} *International Conference on Algebraic Methodology and Software Technology*, Vol. 1101 of *LNCS*, pp. 117–131. Springer.

Kapur, D., & Subramaniam, M. (1996b). Lemma discovery in automating induction. In McRobbie, M., & Slaney, J. (Eds.), *Proc. of the* 13^{th} *International Conference on Automated Deduction*, Vol. 1104 of *LNAI*, pp. 538–552. Springer.

Kapur, D., & Subramaniam, M. (1996c). New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, *16*, 39–78.

Kühler, U., Embacher, C., Eschbach, R., Schumacher, J., Schmidt-Samoa, T., & Sprenger, C. (1998). QuodLibet. System-Dokumentation (German), Fachbereich Informatik, Universität Kaiserslautern, Germany.

Kühler, U., & Wirth, C.-P. (1997). Conditional equational specifications of data types with partial operations for inductive theorem proving. In Comon, H. (Ed.), *Proc. of the* 8^{th} *International Conference on Rewriting Techniques and Applications*, Vol. 1232 of *LNCS*, pp. 38–52. Springer.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley, Reading.

Plaisted, D. A. (1985). Semantic confluence tests and completion methods. *Information and Control*, *65*, 182–215.

Protzen, M. (1994). Lazy generation of induction hypotheses. In Bundy, A. (Ed.), *Proc. of the* 12^{th} *International Conference on Automated Deduction*, Vol. 814 of *LNAI*, pp. 42–56. Springer.

Reddy, U. S. (1990). Term rewriting induction. In Stickel, M. E. (Ed.), *Proc. of the* 10^{th} *International Conference on Automated Deduction*, Vol. 449 of *LNCS*, pp. 162–177.

Schmidt-Samoa, T. (1997). Realisierung einer Taktik-basierten Beweissteuerungskomponente für den induktiven Theorembeweiser QuodLibet. Projektarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, Germany.

Schumacher, J., & Kühler, U. (1997). XQL — a graphical user interface for the inductive theorem prover QuodLibet. Unpublished technical report, Fachbereich Informatik, Universität Kaiserslautern, Germany.

Sprenger, C. (1996). Über die Beweissteuerung des induktiven Theorembeweisers QuodLibet. Diplomarbeit (German), Fachbereich Informatik, Universität Kaiserslautern, Germany.

Walsh, T. (1994). A divergence critic. In Bundy, A. (Ed.), *Proc. of the* 12^{th} *International Conference on Automated Deduction*, Vol. 814 of *LNAI*, pp. 14–28. Springer.

Walther, C. (1988). Argument bounded algorithms as a basis for automated termination proofs. In Lusk, E., & Overbeek, R. (Eds.), *Proc. of the* 9^{th} *International Conference on Automated Deduction*, Vol. 310 of *LNCS*, pp. 601–622. Springer.

Walther, C. (1994). Mathematical induction. In Gabbay, D. M., Hogger, C. J., & Robinson, J. A. (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 2, chap. 13, pp. 127–228. Oxford University Press, Oxford.

Wechler, W. (1992). *Universal Algebra for Computer Scientists.* Springer, Berlin Heidelberg.

Wirsing, M. (1990). Algebraic specification. In van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chap. 13, pp. 675–788. Elsevier Science Publishers B. V., Amsterdam.

Wirth, C.-P. (1995). Syntactic confluence criteria for positive/negative-conditional term rewriting systems. Seki-report SR-95-09, Fachbereich Informatik, Universität Kaiserslautern, Germany.

Wirth, C.-P. (1997). *Positive/Negative-Conditional Equations — A Constructor-Based Framework for Specification and Inductive Theorem Proving*. Verlag Dr. Kovač. PhD Thesis.

Wirth, C.-P., & Gramlich, B. (1994a). A constructor-based approach to positive/negative-conditional equational specifications. *Journal of Symbolic Computation*, *17*, 51–90.

Wirth, C.-P., & Gramlich, B. (1994b). On notions of inductive validity for first-order equational clauses. In Bundy, A. (Ed.), *Proc. of the* 12^{th} *International Conference on Automated Deduction*, Vol. 814 of *LNAI*, pp. 162–176. Springer.

Wirth, C.-P., & Kühler, U. (1995). Inductive theorem proving in theories specified by positive/negative-conditional equations. Seki-report SR-95-15, Fachbereich Informatik, Universität Kaiserslautern, Germany.

Wirth, N. (1971). The programming language Pascal. *Acta Informatica*, *1*, 35–63.

Zhang, H., & Hua, X. (1992). Proving the chinese remainder theorem by the cover set induction. In Kapur, D. (Ed.), *Proc. of the* 11^{th} *International Conference on Automated Deduction*, Vol. 607 of *LNAI*, pp. 431–445. Springer.

Index

sig^C, 20
$\mathcal{T}(sig, V)$, 16
$\mathcal{GT}(sig)$, 16
$\mathcal{GT}(sig^C)$, 20
V^{C}, 20
V^{G}, 20
$\mathcal{T}(sig^C, V^{\mathrm{C}})$, 20
$\mathrm{Var}(t)$, 16
$|t|$, 16
$|t|_x$, 16
$\mathrm{top}(t)$, 16
ε, 16
t/p, 16
$t[u]_p$, 16
$\Gamma[m]$, 205
$\mathrm{Pos}(t)$, 16
$\mathrm{Pos}(\lambda)$, 45
$\mathrm{MinDifPos}(t_1, t_2)$, 47
$\mathrm{MinNonCPos}(t)$, 47
$=_{\mathrm{lit}}$, 45
$t_1 \doteq t_2$, 45
$t_1 \not\doteq t_2$, 45
mgu, 30
$\mathrm{def}(t)$, 20
$\mathrm{eval}^{\mathcal{A}}$, 17
$\mathrm{eval}^{\mathcal{A}}_{\varphi}$, 22, 36
$t^{\mathcal{A}}$, 17
$\ker(h)$, 17
$\mathcal{A}/\!\sim$, 17
$\mathcal{A}^C$, 21
$\models$, 22
$\mathrm{Mod}(spec)$, 22
$\mathrm{DMod}(spec)$, 23
$\mathcal{M}(spec)$, 28
$\longrightarrow$, 17
$\longleftarrow$, 17
$\longleftrightarrow$, 17
$\xrightarrow{+}$, 17
$\xrightarrow{*}$, 17
$\downarrow$, 17
R^C, 24
R^D, 24
$\longrightarrow_{R^C}$, 26
$\longrightarrow_R$, 26
$\mathrm{CP}(R)$, 30
$\mathcal{W}(sig, V)$, 36
$\langle \Gamma\,;\, w \rangle$, 34
$\lesssim$, 17
$<$, 17
$\approx$, 17
A^*, 18
$|w|$, 18
$\leq^{\mathrm{lex}}$, 18
$<^{\mathrm{lex}}$, 18
k_0, 36
$\leq_{\mathcal{A}}$, 38
$\leq^{\mathrm{lex}}_{\mathcal{A}}$, 39
$\precsim_{\mathcal{A}}$, 34, 39
$\mathrm{DefCond}(\mu, \Gamma)$, 58
M_{axiom}, 68
M_{goal}, 68
M_{inf}, 68
$\mathcal{R}$, 70
$\mathcal{S}$, 70
$\mathcal{L}$, 35, 70
$\mathcal{I}$, 35, 70
$\mathrm{OGNd}(P)$, 75
$\mathrm{LNd}(P)$, 75
$\mathrm{IOGNd}(\tilde{\nu}, P)$, 77
$\square$, 78
$\leq_H$, 146, 167
`++`, 133

activate, 153
acyclic, 68

admissible, 26
algebra, 17
AND/OR graph, 72
applicative, 35, 57
arc, 68
    induction hypothesis arc, 70
    lemma arc, 70
    reduction arc, 70
    subgoal arc, 70
`AT` operator, 133
atom, 20, 41
axiom node, 68

C-front, 49
case analysis resulting from literals, 56
choice point, 72
clause, 21
command
    `apply`, 98, 199
    `assert`, 95, 198
    `assign-name`, 99, 199
    `assume`, 97, 198
    `call`, 137, 201
    `compile`, 136, 200
    `declare operators`, 95, 198
    `declare sorts`, 93, 198
    `declare variables`, 93, 198
    `define operator`, 94, 198
    `define sort`, 92, 198
    `delete`, 103, 199
    `display`, 102, 200
    `execute`, 102, 201
    `initialize`, 102, 198
    `load`, 137, 201
    `prove`, 97, 198
    `reset weight`, 103, 200
    `restore IM-state`, 102, 201
    `save IM-state`, 102, 201
    `set current G-node`, 101, 200
    `set current PS-tree`, 101, 200
    `set display-mode`, 138, 200
    `set script-directory`, 102, 200
    `set search-strategy`, 99, 200
    `set weight`, 101, 200
    `unload`, 137, 201
command interpreter, 91, 114
command language, 92, 197
compiler (QML), 123, 136
complement
    of a literal, 21
complementary, 30
condition literal, 21, 60
condition subgoal, 61
condition tree, 149
conditional equation, 21
confluent, 17
congruence, 17
constructor, 20
constructor rule, 25
constructor-consistent extension, 31, 96
contain, 45
context, 47, 52
counterexample, 34
cover set of substitutions, 56, 145, 160, 163
critical pair, 30
current goal node, 97, 99, 101, 117, 137
    `current G-node`, 97, 101, 117, 137
current proof state tree, 97, 101, 137
    `current PS-tree`, 97, 101, 137
cycle, 68
cyclic, 68

data base, 125, 141
data model, 23
data reduct, 21
data type, 20, 128
definedness atom, 20
definedness conditions, 58
definedness subgoal, 155
defining rule, 25
definition scheme, 144, 148
destructor induction, 42
`display-mode`, 138
domain lemma, 152, 155

EBNF, 197
elementary tactic, 130
enumerated type, 129
epimorphism, 17
equation, 20

exception, 131
expandable, 160
expansion, 68, 98
expression, 132, 218
Extended Backus-Naur Form, 197

fail, 130
`FAIL` statement, 131
free algebra, 27
free constructors, 38
function (QML), 130
functionality of QuodLibet, 91

goal, 34
goal node, 68, 97, 117
graphical user interface, 87, 91, 113

head, 60
homomorphism, 17

`IN` operator, 133
induction heuristic, 142
induction hypothesis, 40, 63
induction hypothesis arc, 70
induction lemma, 42, 54, 146, 153, 157
induction order, 39
induction position, 148, 160
induction variable, 37
inductive case analysis, 142, 160
inductive open goal node, 77
inductive theorem, 32
inductively valid, 32
inference machine, 89
inference machine operation, 89
inference node, 68, 117
inference rule
  general form, 35
  safe inference rule, 44
  sound inference rule, 35
  Complementary Literals, 46, 206
  $\neq$-Tautology, 46, 206
  $<$-Tautology, 46, 206
  $=$-Decomposition, 47, 206
  def-Decomposition, 48, 207
  $<$-Decomposition, 49, 207
  Multiple Literals, 50, 207
  $=$-Removal, 50, 208
  $<$-Removal, 50, 208
  $\neq$-Removal, 51, 208
  $\neg$def-Removal, 51, 208
  Constant Rewriting, 52, 208
  $\neq$-Unification, 52, 209
  Constructor Variable Addition, 53, 209
  Tuple $<$-Reduction, 53, 209
  Tuple $=$-Reduction, 53, 209
  $<$-Monotonicity, 54, 210
  $<$-Transitivity, 55, 210
  Substitution Addition, 56, 210
  Literal Addition, 56, 210
  Non-Inductive Subsumption, 58, 211
  Non-Inductive Rewriting, 60, 211
  Applicative Literal Removal, 62, 212
  Inductive Subsumption, 63, 212
  Inductive Rewriting, 64, 213
initial algebra, 17
initial algebra semantics, 20
initial state
  of QuodLibet, 92
  of the inference machine, 102
internal node, 68
invariant for proof state graphs, 77
irreducible, 17
isomorphism, 17

$\mathcal{L}$-free, 73, 74
labeling function, 68
leaf, 68
left-linear, 30
lemma, 10, 153
lemma arc, 70
lemma node, 74
lexicographical order, 18
library routine, 130
linear, 16
list type, 129
literal, 20

`MATCH` operator, 133
measure function, 37
merging (cover set of substitutions), 163
MIL, 89
model, 22

module, 126, 136, 140, 215
  `Database`, 141
  `Inductive-Case-Analyses`, 160
  `Proof-Strategies`, 128, 136, 166
  `Simplification`, 154
monotonic, 18
most general unifier, 29
multiple proof attempts, 70

navigate, 101
node, 68
  axiom node, 68
  goal node, 68, 97, 117
  inference node, 68, 117
  internal node, 68
  start node, 74
non-applicative, 35, 45
normalized, 144

obstruct, 162, 166
occur, 45
open goal node, 74
operand, 132
operator, 133
order atom, 41
order literal, 41
order subgoal, 40, 63, 156

parameter
  reference parameter, 130
  value parameter, 130
partial order, 17
partial proof attempt, 74
path, 68
PCU, 91, 123
position, 16
  minimal, 16
predefined type, 128
procedure, 130
  `activate-lemma`, 153
  `analyze-operator`, 142, 152
  `display-database`, 154
  `display-operator-info`, 154
  `initialize-database`, 141
  `set-weights`, 128, 167
proof attempt, 76

proof control, 87
proof control language, 126
proof control routine, 124
proof control unit, 91, 123
proof graph, 76, 99
proof state forest, 70
proof state graph, 69, 96
proof state tree, 70, 97
proof state tree window, 117, 139
prover object, 132, 219
public routine, 128, 137, 139

QML, 124, 126
  compiler, 123, 136
  expression, 132, 218
  module, 126, 136, 140, 215
  operand, 132
  operator, 133
  routine, 130, 216
  statement, 131, 217
quasi-order, 17
QuodLibet, 85
quotient algebra, 17

recursive, 146
reduction arc, 70
redundant, 50
reference parameter, 130
refutational soundness, 13, 78, 79
rewrite lemma, 154, 158
rewrite rule, 21
rooted tree, 68
routine, 130, 216
$\mathcal{R}/\mathcal{S}$-labeled, 77

safe, 44
satisfy, 22
`search-strategy`, 99
semantic induction order, 41
signature, 16
  constructor signature, 20
software architecture, 89
sort-invariant, 18
soundness
  of an inference rule, 35
  of the framework, 76, 78

  refutational, 13, 78, 79
specification
  admissible, 26
  with constructors, 21
standard data model, 28
start node, 74
state
  of QuodLibet, 92
  of the inference machine, 91, 96
statement, 131, 217
strategy tactic, 125
structured type, 129
subgoal arc, 70
subgraph, 68
substitution, 17
  constructor substitution, 20
  inductive substitution, 20
subsume, 58
subsumption lemma, 154, 158
successor, 68

tactic, 124, 130
  `apply-axiom`, 157
  `apply-ind-hypothesis`, 158
  `apply-lemma`, 158
  `cleanup`, 155
  elementary tactic, 130
  `expand-operator`, 166
  `ind-case-analysis`, 160
  `prove-def-subgoal`, 156
  `prove-order-subgoal`, 157
  `prove-taut-lit`, 134
  `prove-tautology`, 134, 155
  `restricted-strategy`, 128, 171
  `simplify-goal`, 159
  `standard-strategy`, 128, 166, 168
terminating, 17
termination heuristic, 146
termination witness, 146
tree, 68
  proof state tree, 70
`TRY` statement, 131
type, 128, 216
  enumerated type, 129
  list type, 129
  predefined type, 128
  structured type, 129

unifier, 29

valid, 22
valuation, 22
value parameter, 130
variable, 16
  constructor variable, 20
  general variable, 20

weakly normal, 30
weight, 34, 36
weight variable, 97, 102, 166
well-founded, 17

XQuodLibet, 89, 113

Lebenslauf

Name:	Ulrich Kühler	
Geburtsdatum:	31.05.1964	
Geburtsort:	Castrop-Rauxel	
Familienstand:	verheiratet, zwei Kinder	
Staatsangehörigkeit:	deutsch	
Schulausbildung:	08/1970 – 03/1972	Richtsbergschule in Marburg
	03/1972 – 02/1974	Windthorst-Schule in Wuppertal
	02/1974 – 06/1974	Grönebergschule in Melle
	08/1974 – 07/1980	Gymnasium Melle
	08/1980 – 06/1981	Eisenhower Senior High School in Washington, Michigan, USA
	08/1981 – 06/1984	Gymnasium Melle
Schulabschluss:	30.06.1984	Allgemeine Hochschulreife
Grundwehrdienst:	07/1984 – 09/1985	
Studium:	10/1985 – 11/1991	Informatik mit Nebenfach Mathematik an der Universität Kaiserslautern
Studienabschluss:	14.11.1991	Diplom-Informatiker
Berufstätigkeit:	11/1991 – 09/1998	Wissenschaftlicher Mitarbeiter im Fachbereich Informatik an der Universität Kaiserslautern