$_3T^AP$

The Many-Valued Theorem-Prover

Reiner Hähnle
Bernhard Beckert
Stefan Gerberding

3rd Edition
September 1994

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems

Abstract

This is the $_3T^AP$ handbook. $_3T^AP$ is a many-valued tableau-based theorem prover developed at the University of Karlsruhe.

The handbook serves a triple purpose: first, it documents the history and development of the prover $_3T^AP$; second, it provides a user's manual, and third it is intended as a reference manual for future developers, including porting hints.

This version of the handbook describes $_3T^AP$ Version 3.0 as of September 30, 1994.

Authors' Address

Bernhard Beckert, Reiner Hähnle:

Institute for Logic, Complexity and Deduction Systems
Department of Computer Science
University of Karlsruhe
Kaiserstraße 12
76128 Karlsruhe
Germany

Email: `beckert@ira.uka.de`, `reiner@ira.uka.de`
WWW: `http://i12www.ira.uka.de/`

Stefan Gerberding:

Institute for Programme- and Inference Systems
Department of Computer Science
Technical University of Darmstadt
Alexanderstraße 10
64283 Darmstadt
Germany

Email: `stefan@inferenzsysteme.informatik.th-darmstadt.de`
WWW: `http://kirmes.inferenzsysteme.informatik.th-darmstadt.de/~stefan`

Contents

Preface

This is the $_{3}T^{A}P$ handbook. $_{3}T^{A}P$ (Three-valued Tableau-based Automated Theorem Prover) is a many-valued tableau-based theorem prover developed at the University of Karlsruhe. Despite its name, $_{3}T^{A}P$ is capable of handling arbitrary finitely-valued first-order logics (and, of course, classical two-valued logic as well).

The handbook serves a triple purpose: first, it documents the history and development of the prover $_{3}T^{A}P$; second, it provides a user's manual, and third it is intended as a reference manual for future developers, including porting hints.

The decision to include all this in a single document was based on the observation that all three parts have a considerable amount of intersection and we can minimize the costs for preparation and printing this way. We hope to deviate the main disadvantage of this form of presentation, the danger of becoming unhandy, with a clear structure of the chapters, generously placed cross references and a subject index.

In Chapter 1 we give a detailed description of $_{3}T^{A}P$'s history and that of the projects as part of which $_{3}T^{A}P$ was developed and is still being maintained, and we summarize the ensuing problems and how they were solved. In Chapter 2 we provide the theoretical background which is needed for a deepened understanding of the way the prover works. In particular we discuss the calculus which is used in the prover. In Chapter 3 the formal syntax of the input to the prover is given and discussed. Chapter 4 provides a general overview of $_{3}T^{A}P$'s architecture. Its modularization and the interplay between the modules are discussed. The modules are described in detail in Chapter 5. Together with the documentation in the source code itself these descriptions aim to give a future implementor enough information for successfully making changes to $_{3}T^{A}P$. Several utility programs which are not part of the source code of $_{3}T^{A}P$ itself and which are therefore not required for running it are described in Chapter 6. These concern exclusively the inspection of formal tableau proofs generated by $_{3}T^{A}P$. Chapter 7 presents a tutorial for using $_{3}T^{A}P$ in form of a sample session in the course of which all relevant features are used and discussed. Chapter 8 is dedicated to the evaluation of $_{3}T^{A}P$. Part of the $_{3}T^{A}P$-package is a batch of test problems for classical and many-valued logic on the propositional as well as on the first-order level. We explain how to use these test problems, give statistical figures for the latest version and discuss evidence of these tests. $_{3}T^{A}P$ is capable of dealing with arbitrary logics with finitely many truth values. Chapter 9 lists the changes that have to be made in order to adapt $_{3}T^{A}P$ to a new logic. Appendixes A and B serve as references for available commands and switches, respectively. Finally, Appendix C shows how to install $_{3}T^{A}P$ on a computer and what has to be changed, when it is to be ported to a different configuration than the supported ones.[1]

The aforementioned triple purpose of this report implies that not all parts are equally relevant for all potential readers. Therefore, we make a suggestion which readers should read what chapters:

The reader who quickly wants to get a general impression of the prover $_{3}T^{A}P$ should read Chapters 1, 4, 6.1, 8.1.1, and 8.1.2. If he or she is interested in the theoretical aspects, Chapter 2 is adequate additional reading, while readers who want to get a feeling of how $_{3}T^{A}P$ behaves in reality should also consult Chapter 7.

[1] These are currently: Quintus Prolog 3.0/3.1 and SICStus Prolog 2.1 on SUN Sparc under SunOS 4.1.x.

The reader who wants to use $_{3}T^{A}P$ should read Chapters 1, 3, 6.1, 6.2, 6.4, 6.5, 7, 8.1, 8.2 and Appendices A, B. If he or she wants to use the many-valued version, Chapters 2 and 9 should be at least skimmed.

The reader who wishes to change or port $_{3}T^{A}P$ should read anything recommended for the user plus Chapter 4 and the sections of Chapters 5, 6, 9 and Appendix C that match his purposes. Since many of the modules are closely interacting, however, a complete reading of the whole report is recommended.

The reader who wants to install $_{3}T^{A}P$ may read—in addition to Appendix C—Sections 3.2, 3.4 and 5.15, and for testing Chapter 7 or Chapter 8.

The reader who is particularly interested in the handling of equality should read Sections 2.6, 5.5.7, 5.13, and 5.14.5.

To our best knowledge $_{3}T^{A}P$ is the first and only automated theorem prover capable of dealing with arbitrary many-valued logics. Moreover, the statistical figures in Chapter 8 suggest that its two-valued version (and therefore also the many-valued version, for reasons that become clear in Chapter 2) does perform not too bad if compared to conventional theorem provers.

This version of the handbook describes $_{3}T^{A}P$ Version 3.0 as of September 30, 1994.

What's New in Version 3.0?

The major changes that have been made (as compared to Version 2.1) are:

- $_{3}T^{A}P$ now uses a completion-based method for equality handling (instead of the old method, that was based on computing equivalence classes).[2]
- $_{3}T^{A}P$ has been ported to SICStus Prolog. It now runs under SICStus Prolog 2.1 (and Quintus Prolog 3.0/3.1).
- The compiler, that translates knowledge bases into $_{3}T^{A}P$'s internal representation, has been re-implemented.

Besides that, there have been a lot of small improvements.

Sections Taken from Other Publications

With the authors' permission, substantial parts of Chapter 2 have been taken from (Hähnle, 1992c), Section 2.5 has been taken from (Beckert & Hähnle, 1992; Beckert & Posegga, 1994b), and Section 2.6 from (Beckert, 1994b) and (Beckert, 1994a).

Acknowledgments

As Included in the First Edition

Before diving into the facts it is my duty and my pleasure to thank all people who have been involved in or helped with TCG Karlsruhe in some way or another:

Stefan Bayerl, Toni Bollinger, Sven Dörr, Thomas Kropf, Sven Lorenz, Sabine Lückehe, Markus Mock, Martin Müller, Prof. Daniele Mundici, Prof. Neil Murray, Udo Pletat, Joachim Posegga,

[2] The new method for equality handling can be used stand-alone; see Section C.7.

Klaus Ries, Prof. Wolfgang Schönfeld, Gisela Schöpke, Gernot Stenz, Andreas Völter, Wolfgang Wernecke. Extra thanks go to Martin Kreidler for writing the section on dissolution.

Special thanks go to Bernhard Beckert, Stefan Gerberding and Werner Kernig who constantly worked as “Hiwis” for TCG with great expertise and devotion and who are also co-authors of this report. Without their help the project simply could not have been done.

A extra-special thank you goes to Prof. Peter Schmitt, the initiator of TCG Karlsruhe. In the early stages of the project he gave the necessary guidelines, in the later stages the necessary freedom and invaluable comments at any time. It is mainly his achievement that many-valued theorem proving has lost the obscurity it had for most people three years ago.

Reiner Hähnle
Karlsruhe, May 1992

As Included in the Second Edition

We would like to thank the following people for testing $_3T^AP$, pointing out errors, suggesting improvements or helping in other ways to develop the new version: Stefan Gerberding, Ortrun Ibens, Bertram Ludäscher, Martin Müller, Prof. Peter Schmitt.

Reiner Hähnle, Bernhard Beckert
Karlsruhe, February 1993

For the Third Edition

Again, a lot of people helped improving $_3T^AP$. We would like to thank the following people for testing $_3T^AP$, pointing out errors, or suggesting improvements: Ulrich Böck, Uwe Egly, Stefan Gerberding, Peter Oel, Christian Pape, Prof. Uwe Petermann, Joachim Posegga, and Prof. Peter Schmitt.

Special thanks go to Ortrun Ibens, who works as “Hiwi” for the project. She implemented the new compiler, ported $_3T^AP$ to SICStus Prolog, and helped improving $_3T^AP$ in various other ways.

Bernhard Beckert, Reiner Hähnle
Karlsruhe, September 1994

1 Introduction

1.1 History and Development of $_3T^AP$

1.1.1 The TCG Project

In June 1989 IBM Germany launched ILFA[1], a joint project together with the Universities of Duisburg and Karlsruhe with the aim of developing an integrated environment that provides tools for knowledge-based inference systems. In July 1990 the subprojects of IKBS at IBM Germany in Heidelberg and University of Karlsruhe were separated from the mainstream and labelled TCG[2].

The task of the Karlsruhe part of TCG was to develop a many-valued theorem proving system, suitable at least for a certain three-valued logic that occurred in connection with natural language processing (Fenstad *et al.*, 1985; Schmitt, 1989). The area of natural language processing was the initial motivation for building a many-valued theorem prover, but it has been later given up in favour of hardware verification. Since no serious implementations of a many-valued theorem prover have ever been reported, the first goal of the project was to lay the theoretical foundations for a way to do that efficiently. The project tasks (in updated form from July 1990) were thus:

1. Specification of the theoretical foundations of a three-valued inference engine.
2. Specification of the theorem prover.
3. Implementation of a prototype.
4. Integration of the prototype with TC[3] and LEU[4]
5. Evaluation of the prototype (a) on module level and (b) within LEU.
6. Specification of the theoretical foundations of an inference engine for a temporal logic.

The basic tasks 1, 2, 3, 4, 5(a) have been fully accomplished. Tasks 5(b) and 6 proved to be not fully accomplishable under the given organizational and time constraints. The input language specification, the first version of the compiler[5], the index generator and most of the integration work with TC and LEU was done by TCG Heidelberg. The other parts were done in Karlsruhe.

Another measure of the project's success is the scientific progress that was made during its lifetime. Five scientific papers, three technical reports, five Studienarbeiten, two Diplomarbeiten and one dissertation have been produced in connection with it (see Section 1.4). The theoretical

[1] **I**ntegrated **L**ogical **F**unctions for **A**dvanced Applications.
[2] **T**ableau **C**alculus mit **G**leichheit.
[3] A system that has been developed at IKBS in Heidelberg
[4] **L**ogik **E**ntwicklungs**u**mgebung.
[5] Later, in 1994, the compiler was re-implemented in Karlsruhe.

advances made during the course of the project reach far beyond many-valued logics and will be further developed at the University of Karlsruhe and other places after the end of TCG.

It is our opinion that both the fields of many-valued logic and of tableau-based theorem proving have received interesting stimuli by TCG.

The people who were chiefly concerned with design and implementation in Heidelberg are Wolfgang Schönfeld and Wolfgang Wernecke for TC and in Karlsruhe Reiner Hähnle for $_3T^AP$. Peter Schmitt supervised TCG Karlsruhe.

1.1.2 The DFG Schwerpunktprogramm "Deduction Systems"

Since June 1992, after the end of the TCG project, $_3T^AP$ has been maintained and is still being improved as part of a new project at the University of Karlsruhe, that is funded by the *Deutsche Forschungsgemeinschaft* (DFG) as part of the *Schwerpunktprogramm* "Deduction Systems". Objective of this project is to refine proof methods based on semantic tableaux, i.e., to add new features and to increase their efficiency. In this context $_3T^AP$ is used as a platform for experiments and for testing new methods.

The people mainly involved with the project are Reiner Hähnle (executive until July 1993), Bernhard Beckert (executive since July 1993), Joachim Posegga, and Prof. Peter Schmitt (supervisor).

1.1.3 Chronology of $_3T^AP$'s Development

In the following table we give a chronological history of $_3T^AP$'s development.

June 1989	TCG Karlsruhe was launched, then under the name of ILFA.
October 1989	Since this date several possible scenarios for an application oriented evaluation were considered.
November 1989	Overall design of $_3T^AP$ completed.
December 1989	The theoretical basis of $_3T^AP$ emerged. The concept of sets-as-signs was developed and later published (Hähnle, 1990a; Hähnle, 1990b; Hähnle, 1991). It became clear that the restriction to three truth values is unnecessary and it was dropped henceforth.
February 1990	Start of implementation of uncritical modules.
May 1990	Theoretical foundations completed.
June 1990	Specification completed.
July 1991	Separation from ILFA. As a consequence, the prover TC developed in Heidelberg and $_3T^AP$ had to be merged.
September 1990	Theoretical foundations and specification published as (Hähnle, 1990a).
October 1990	International presentation of the theoretical concepts of $_3T^AP$ at CSL'90; published in (Hähnle, 1990b).
January 1991	First prototype ready for classical first-order logic.

February 1991	Specification of common core with TC and interface to LEU. It was decided to take the input syntax from TC which is a subset of the LEU syntax as a common basis. Compiler and preprocessing modules are taken from TC while the prover and internal data structures come from $_{3}T^{A}P$ which is also the name of the common system. Since TC was capable of handling order-sorted logic $_{3}T^{A}P$ was extended accordingly.
March 1991	Many-Valued prototype ready. This prototype is evaluated in a scenario from decision support. The results are documented in (Schöpke, 1991).
April 1991	Integration of TC and $_{3}T^{A}P$ completed.
May 1991	International presentation of further theoretical results; published in (Hähnle, 1991).
July 1991	It was decided that the main evaluation of $_{3}T^{A}P$ should be done in the area of hardware verification and as a stand alone system, i.e. not as a part of LEU.[6]
July 1991	Hardware scenario is prepared.
July 1991	Two-Valued equality handling implemented. It is documented in (Beckert, 1992). The approach marks a substantial advance in tableau-based equality proving and will be internationally presented in (Beckert & Hähnle, 1992).
December 1991	Successful evaluation of $_{3}T^{A}P$ with a three-valued first-order logic in the domain of interval arithmetic; documented in (Gerberding, 1991).
March 1992	$_{3}T^{A}P$ is integrated in LEU.
March 1992	Significant new developments of the tableau framework reaching beyond the current $_{3}T^{A}P$ implementation are presented on an international workshop (Hähnle, 1992a). These imply the possibility of building an efficient prover for temporal logics.
April 1992	As a possible means of enhancing the performance of $_{3}T^{A}P$, the dissolution rule is added to the two-valued version and tests are performed; documented in (Kreidler, 1992).
May 1992	Successful evaluation of $_{3}T^{A}P$ with a seven-valued propositional logic in the domain of hardware verification; documented in (Kernig, 1992).
May 1992	Compilation of this report and handing over the final version of $_{3}T^{A}P$ to IBM. End of the TCG project.
June 1992	The DFG project begins.
February 1993	The 2nd edition of the $_{3}T^{A}P$ handbook is completed.
June 1993	The new completion-based equality handling replaces the old method, that was based on computing equivalence classes.
December 1993	$_{3}T^{A}P$ is ported to SICStus Prolog.
April 1994	The compiler module that translates knowledge bases into $_{3}T^{A}P$'s internal representation is re-implemented.
August 1994	Completion of $_{3}T^{A}P$ Version 3.0 and of this version of the handbook.

[6] There are two reasons for this decision: first, the interest in linguistic applications had ceased at IKBS; since LEU is part of a natural language processing system, it would have made no longer sense to evaluate $_{3}T^{A}P$ within LEU. Second, the machines provided by IBM for the Karlsruhe TCG project were too small to run LEU.

1.2 $_3T^AP$'s Main Features

The implementation language of $_3T^AP$ is Prolog (there are versions for Quintus Prolog and for SICStus Prolog) with a small amount of portable C. Parts of $_3T^AP$'s compiler module and of the utilities for visualizing proofs are written using the Unix tools Lex and Yacc (resp. Flex and Bison).

It is easy to install $_3T^AP$ on a Unix machine with Quintus Prolog Version 3.0 (or later) or SICStus Version 2.1 (or later) and a C compiler available. To achieve acceptable performance, at least 8MB main memory and 20MB swap space should be configured.

The design of $_3T^AP$ is as modular as possible. Thus, it should be easy to port it to other architectures than the one described above and to add new features.

$_3T^AP$ Version 3.0 as of September 30, 1994 includes the following features:

- Full two-valued first-order logic with equality and sorted terms.
- Many-valued first-order logic with quasi-classical quantifiers. The prover is adaptable to any finitely-valued logic involving arbitrary connectives within a few hours, provided the truth tables of the connectives are given. The process is described in detail in Chapter 9.

  Sorted terms and two-valued equality are still available with many-valued logics.
- Various strategies that may shorten proofs such as lemma generation are implemented and may optionally be switched on in all versions.
- In the two-valued version a restricted version of the dissolution inference rule of Murray & Rosenthal (Murray & Rosenthal, 1987) is implemented and may optionally be switched on.

$_3T^AP$ is able to prove benchmark problems for (two-valued) theorem provers in reasonable time, see Chapter 8. Response times for problems which are not too difficult are typically under 500ms.

The user interface of $_3T^AP$ is the Prolog shell enriched with certain predicates for proving theorems, loading databases etc. This has the advantage that the user can quickly write his own predicates on top of these to accomplish specialized tasks. Moreover, it is very easy to integrate $_3T^AP$ into any existing system via its interface predicates.[7]

The syntax of databases is based on that of first-order predicate logic in a very natural notation. No normal form is required. In particular, equalities may occur in arbitrary places within a formula, see Chapter 3 for examples and details.

1.3 Theoretical Advances

Starting point for TCG Karlsruhe has been the requirement to use analytic tableaux as a general framework and a theoretical paper by Carnielli (Carnielli, 1987) describing a way to axiomatize every finitely-valued logic with analytic tableaux. This approach proved to be intractable in practice even for very small examples.

The solution was a consequent generalization of the sign language in analytic tableaux. It has the advantage that a kind of *semantic structure sharing* can automatically be achieved as depicted in Figure 1.1. For the technical details we refer to Chapter 2. The result was first published in

[7] Currently, an X-Windows based interface is under development. It will be part of $_3T^AP$'s next release.

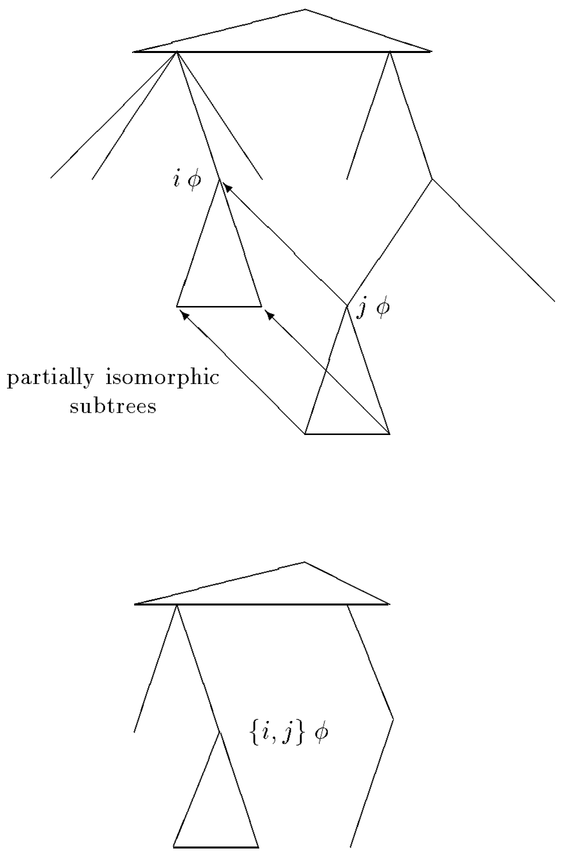

Figure 1.1: Schema of semantic structure sharing using sets-as-signs.

(Hähnle, 1990b). The significance of the technique is backed up by the fact that in the meantime it has (independently) been rediscovered for special cases by other researchers (Doherty, 1991). We will call the approach of (Hähnle, 1990b) which is crucial for $_3T^AP$'s efficiency the **sets-as-signs** approach to many-valued reasoning, since it operates by using subsets of the set of truth values as signs in tableaux.

Starting from the sets-as-signs approach we were able to identify many-valued logics that have particularly simple tableau proof systems if Smullyan's uniform notation (Smullyan, 1968) is adapted. Moreover, we were able to give very simple quantifier rules for the many-valued generalizations of $\forall$ and $\exists$ (Hähnle, 1991) under mild restrictions.

As a byproduct of the project a new liberal version of the classical δ-rule for first-order tableaux which was first mentioned in (Schmitt, 1987) could be proven to be sound (Hähnle & Schmitt, 1993). Later, an even more liberalized version was developed (Beckert *et al.*, 1993).

The investigations made for implementation of equality (Beckert, 1992) have also proved to be fruitful and were internationally acknowledged (Beckert & Hähnle, 1992; Beckert, 1994b). $_3T^AP$ is the first working tableau-based prover that is able to solve problems with equality beyond the scope of textbook examples.

A further generalization of the sets-as-signs approach (Hähnle, 1992a) resulted in a new translation from (many-valued) deduction problems into the domain of mixed integer programming. Immediate consequences are simple NP-containment proofs for various many-valued logics that have been difficult before and, for the first time, the possibility to efficiently perform theorem proving in *infinitely-valued* logics. Other consequences of this new technique whose presentation is well beyond the scope of this report cannot fully be foreseen yet. They will be investigated in the future.

Finally, the evaluation scenario within hardware verification (Kernig, 1992; Beckert *et al.*, 1994) for the first time accomplished formal verification of switch-level specifications with deductive methods. These have up to now been handled by mere simulation tools.

A multitude of statistical figures will be given in Chapter 8. Here, we provide some figures to back up our claim that an efficiency jump in many-valued theorem proving could be achieved using the sets-as-signs technique. In Table 1.1 we give numbers of closed branches (these resemble approximately the size of the proof) and run times for a couple of problems using the naïve approach and the sets-as-signs approach. The second column indicates whether the problem is satisfiable.

problem	sat	naïve		sets-as-signs	
		Time*[s]*	Branches	Time*[s]*	Branches
`thhornor1`	Y	8.00	1334	0.27	36
`thvernor1`	Y	5.13	806	0.15	17
`thvernor3`	N	25.47	4487	2.93	518
`Lem. 5.1`	Y	1.18	206	0.17	7
`Fig. 5.17`	Y	∞	∞	1.37	36

Table 1.1: Comparing naïve and sets-as-signs approaches.

The first three problems are taken from (Kernig, 1992) and are about hardware verification. They are formulated in a propositional seven-valued logic. The other problems from (Gerberding, 1991) are about interval arithmetic and formulated in a three-valued first-order logic. The gain of the sets-as-signs approach is in both cases considerable. The last problem could only be solved with sets-as-signs.

$_3T^AP$ can be seen—to a certain extent—as the parent of the leanT^AP prover (Beckert & Posegga, 1994c; Beckert & Posegga, 1994b). leanT^AP is an instance of *lean deduction*. It is written in Prolog and implements a complete and sound theorem prover for classical first-order logic based on free-variable semantic tableaux. The unique thing about leanT^AP is that it is probably the smallest theorem prover around. The idea of lean deduction is to achieve maximal efficiency from minimal means. Every possible effort is made to eliminate overhead; based on experience in implementing (complex) deduction systems—namely $_3T^AP$—, only the most important and efficient techniques and methods are implemented.

1.4 Documentation

We summarize the papers and reports written in connection with $_3T^AP$, or as part of the TCG project or the DFG Schwerpunkt, and give their type and purpose.

Reference	Type	Purpose/Remarks
(Hähnle, 1990a)	IWBS Report	Overall design and coarse specification of $_3T^AP$. Sketch of sets-as-signs concept. Partially outdated now. In German.
(Hähnle, 1990b)	In proceedings	Sets-as-signs concept in full detail.
(Mock, 1990)	Studienarbeit	Investigation into and implementation of many-valued function minimization algorithms, cf. (Hähnle, 1992c). In German.
(Kernig, 1990)	Studienarbeit	Proving theorems about bilattices with OTTER. In German.
(Gerberding, 1990)	Studienarbeit	Exploring the C interface of Quintus Prolog. In German.
(Hähnle, 1991)	In proceedings	Uniform notation for many-valued logics and many-valued first-order logic.
(Wernecke, 1991)	Internal paper	TCG input language specification of integrated system.
(Beckert, 1991)	Studienarbeit	Building equality into $_3T^AP$. In German.
(Gerberding, 1991)	Diploma thesis	Axiomatizing interval arithmetic with multiple-valued logic and evaluating $_3T^AP$ with it. In German.
(Beckert & Hähnle, 1992)	In proceedings	Building equality into a tableau-based prover.
(Schöpke, 1991)	IWBS Report	Study of potential application areas for a many-valued theorem prover.
(Beckert, 1992)	IWBS Report	Building equality into $_3T^AP$. In German.
(Kernig, 1992)	Diploma thesis	Using many-valued logic for hardware verification and evaluating $_3T^AP$ with it. In German.
(Kreidler, 1992)	Studienarbeit	Building dissolution into $_3T^AP$. In German.
(Hähnle, 1992a)	In proceedings	Short version of (Hähnle, 1992b).
(Hähnle, 1992b)	In proceedings	Translation from semantic tableaux to integer programming.
(Hähnle, 1992c)	PhD thesis	Theory of multiple-valued tableau-based theorem proving.
(Hähnle & Schmitt, 1993)	Article	Liberalized δ-rule in classical tableaux. To appear.
(Hähnle & Kernig, 1993)	In proceedings	Using many-valued logic for hardware verification.
(Beckert *et al.*, 1993)	In proceedings	The even more liberalized δ-rule in free variable tableaux.
(Hähnle, 1993a)	Book	Monograph on automated proof search in multiple-valued logics. Includes an extensive bibliography.
(Hähnle, 1993c)	In proceedings	Short normal forms for arbitrary finitely-valued logics.

Reference	Type	Purpose/Remarks
(Beckert, 1993a)	In proceedings	Completion-based handling of equality in semantic tableaux.
(Hähnle, 1993b)	In proceedings	Efficient deduction in many-valued logics.
(Beckert, 1993b)	Diploma thesis	Completion-based handling of equality in semantic tableaux. In German.
(Hähnle, 1994a)	In proceedings	Efficient deduction in many-valued logics.
(Beckert & Posegga, 1994c)	In proceedings	Description of the leanT^AP prover. Short version of (Beckert & Posegga, 1994b).
(Beckert, 1994b)	In proceedings	A completion-based method for solving mixed E-unification problems.
(Hähnle, 1994b)	Article	Short conjunctive normal forms in finitely-valued logics. To appear.
(Beckert, 1994a)	In proceedings	Overview: Adding equality to semantic tableaux.
(Beckert & Posegga, 1994a)	In proceedings	Position paper on *lean deduction.*
(Beckert, 1994c)	In proceedings	Using E-unification to handle equality in universal formula semantic tableaux.
(Beckert *et al.*, 1994)	In proceedings	Description of the concept of “anti-links”.
(Beckert & Posegga, 1994b)	Article	Description of the leanT^AP prover. To appear.

Remark 1.1 (Wernecke, 1991) *and* (Schöpke, 1991) *were contributed by TCG Heidelberg. The latter paper was written during a guest scholarship of Dr. Schöpke at IBM Heidelberg.*

1.5 Getting $_3T^AP$ and the Documentation

1.5.1 Copyright

$_3T^AP$ Version 3.0 is copyrighted © 1994 by the University of Karlsruhe, Institute for Logic, Complexity and Deduction Systems.

Permission to use, copy, and distribute this software and its documentation is hereby granted, subject to the following conditions:

1. Permission to modify $_3T^AP$ is granted, but not the right to distribute the modified code.
2. If $_3T^AP$ is distributed—in whole or part—, the above copyright notice must appear in all copies, and both that copyright notice and this permission notice must appear in supporting documentation.
3. Distributors must not demand a fee for distributing $_3T^AP$.
4. $_3T^AP$ must not be used for commercial purposes without the written consent of the authors.

$_3T^AP$ is provided “as is” without express or implied warranty.

1.5.2 How to Get $_3T^AP$ via the Word Wide Web

The easiest (and most fashionable) way to get $_3T^AP$’s source code and most of the documents listed in Section 1.4 is opening the page

```
http://i12www.ira.uka.de/~threetap
```

on the World Wide Web (WWW). With this document you can retrieve the program and the documentation online.

Once you have got hold of the file `threetap.tar.gz`, which is a compressed TAR file, and contains both the source code and the $_{3}T^{A}P$ Handbook, use the shell command

```
gunzip threetap.tar.gz
```

to uncompress `threetap.tar.gz`. Then unpack the TAR file `threetap.tar` using the command

```
tar -xf threetap.tar
```

A directory `threetap` containing the $_{3}T^{A}P$ source code and the Handbook will be generated.

For those who have no access to the WWW, we describe the access by anonymous FTP in the following section. If you should not have FTP access, either, contact the authors (see Section 1.5.4).

1.5.3 How to Get $_{3}T^{A}P$ via Anonymous FTP

$_{3}T^{A}P$'s source code and the documentation is also available via anonymous FTP on Internet from

```
sonja.ira.uka.de   [129.13.31.3]
```

Open this host with an FTP program, log in as "anonymous" and type your e-mail address as password. Change (`cd`) to the directory `pub/threetap` and switch to binary file transfer mode. This is usually done by typing "binary" in your FTP program.

Then get the file `threetap.tar.gz`, and proceed to uncompress and unpack this file as described in the previous section.

1.5.4 Please Contact Us

If you are interested in getting the $_{3}T^{A}P$ source code or some of the documents, and you do not have access to the WWW or FTP, if you have any questions or suggestions concerning $_{3}T^{A}P$, or if you wish to be on an e-mail list to receive updates, bug fixes, information on new releases, etc., then please contact us (preferably by e-mail). We appreciate feedback! The contact address is:

Institute for Logic, Complexity and Deduction Systems
Department of Computer Science
University of Karlsruhe
Kaiserstraße 12
76128 Karlsruhe
Germany

Email: `beckert@ira.uka.de`, `reiner@ira.uka.de`
WWW: `http://i12www.ira.uka.de/`

2 Theoretical Background

2.1 Analytic Tableaux

Semantic (or analytic) tableaux have been introduced by E. W. Beth (Beth, 1986) and K. J. J. Hintikka (Hintikka, 1955) in the 1950s, its ancestors being Gentzen systems. R. Smullyan (Smullyan, 1968) gave a particular elegant version of tableaux which increased their popularity largely and most tableaux systems used today are based on the formulation he gave. Tableaux systems come in two versions, namely signed and unsigned, from which we will always be using the former.

In the classical case our set of signs (sometimes also called prefixes) will be $\{\mathsf{F}, \mathsf{T}\}$ where F of course corresponds to the truth value 0 and T to 1.

Definition 2.1 (Signed Formula) *A signed formula is a string of the form* $\mathsf{S}\,\phi$*, where* ϕ *is a (propositional or first-order) formula and* S *is either* F *or* T*. If* $\mathbf{L}$ *is the set of formulae in a logic, the set of signed formulae will be denoted with* $\mathbf{L}^*$*.*

Following Smullyan we may divide the set of signed formulae into four classes: α for propositional formulae of conjunctive type, β for propositional formulae of disjunctive type, γ for quantified formulae of universal type and finally δ for quantified formulae of existential type. Smullyan called this *uniform notation*. It simplifies presentation and proofs considerably.

$\dfrac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}}$	$\dfrac{\beta}{\beta_1 \mid \beta_2}$	$\dfrac{\gamma}{\gamma_1(t)}$	$\dfrac{\delta}{\delta_1(c)}$
		where t is an arbitrary term.	where c is a Skolem constant that is not occurring on the current branch.

Table 2.1: Tableaux rule schemas for different formula types.

This classification is motivated by the *tableau expansion rules* which are associated with each signed formula. The rules characterize the assertion of a truth value (corresponding to its sign) to a formula by means of asserting truth values to its direct subformulae. For example, $\mathsf{T}\,(\phi \wedge \psi)$ holds if and only if $\mathsf{T}\,\phi$ and $\mathsf{T}\,\psi$ hold. In Table 2.1 the rule schemes for the various combinations of signs and formula types are given schematically. Premises and conclusions are separated by a horizontal bar, while vertical bars in the conclusion denote different *extensions* which are to be thought as disjunctions. In Table 2.2 the correspondence between formulae and formula types is shown.

α	α_1	α_2
$\mathsf{T}\,(\phi \wedge \psi)$	$\mathsf{T}\,\phi$	$\mathsf{T}\,\psi$
$\mathsf{F}\,(\phi \vee \psi)$	$\mathsf{F}\,\phi$	$\mathsf{F}\,\psi$
$\mathsf{T}\,\neg\phi$	$\mathsf{F}\,\phi$	$\mathsf{F}\,\phi$
$\mathsf{F}\,\neg\phi$	$\mathsf{T}\,\phi$	$\mathsf{T}\,\phi$

β	β_1	β_2
$\mathsf{T}\,(\phi \vee \psi)$	$\mathsf{T}\,\phi$	$\mathsf{T}\,\psi$
$\mathsf{F}\,(\phi \wedge \psi)$	$\mathsf{F}\,\phi$	$\mathsf{F}\,\psi$

γ	$\gamma_1(t)$
$\mathsf{T}\,(\forall x)\phi(x)$	$\mathsf{T}\,\phi(t)$
$\mathsf{F}\,(\exists x)\phi(x)$	$\mathsf{F}\,\phi(t)$

δ	$\delta_1(c)$
$\mathsf{F}\,(\forall x)\phi(x)$	$\mathsf{T}\,\phi(c)$
$\mathsf{T}\,(\exists x)\phi(x)$	$\mathsf{F}\,\phi(c)$

Table 2.2: Correspondence between rule types and formulae.

For our purposes it is sufficient to visualize a tableau proof as a finite labelled binary tree, whose node labels are signed formulae, constructed as follows:

Definition 2.2 (Classical Tableaux) *Let $\mathbf{L}^*$ be a language of signed formulae. The set $\mathcal{T}(\mathbf{L}^*)$ of all tableaux over $\mathbf{L}^*$ is defined as the set of trees that can be constructed by finitely many applications of the following rules:*

(T1) A finite linear tree whose nodes are signed formulae taken from a set $\Phi \subseteq \mathbf{L}^*$ is a tableau over $\mathbf{L}^*$.

(T2) If $\mathbf{T}$ is a tableau over $\mathbf{L}^*$ and ϕ is a node label from $\mathbf{T}$ then a new tableau $\mathbf{T}'$ is constructed by extending all branches of $\mathbf{T}$ that contain ϕ by as many new linear subtrees as the rule[1] *corresponding to ϕ has extensions, the nodes of the new subtrees being labelled with the formulae in the extensions.*

If $\mathbf{T}$ is a tableau and Φ is the set in step (T1) above, then $\mathbf{T}$ will also be called a tableau for Φ.

Definition 2.3 (Branch) *Let $\mathbf{T}$ be a tableau. A branch $\mathbf{B}_{\mathbf{T}}$ of $\mathbf{T}$ is a maximal path in $\mathbf{T}$.*

Usually, when no confusion can arise, we omit the subscript from $\mathbf{B}_{\mathbf{T}}$. Sometimes, when we speak of a branch $\mathbf{B}$, we actually mean the set of node labels (signed formulae) on $\mathbf{B}$, but we still use the symbol $\mathbf{B}$.

Definition 2.4 (Complementary Signs and Formulae) *Two signs $\mathsf{S}_1, \mathsf{S}_2$ are complementary iff $\mathsf{S}_1 \neq \mathsf{S}_2$. Let $\mathsf{S}_1\phi, \mathsf{S}_2\psi$ be two formulae on a tableau branch $\mathbf{B}$. They are called complementary formulae iff $\mathsf{S}_1, \mathsf{S}_2$ are complementary signs and $\phi = \psi$.*

Definition 2.5 (Closed and Open Branch) *A tableau branch is closed iff it contains a pair of complementary formulae. Otherwise it is called open.*

[1] It is obtained by looking up the subformulae corresponding to ϕ in Table 2.2 and instantiating the matching rule schema in Table 2.1.

To prove tautologyhood of a formula ϕ we begin with a tree whose single node is labelled by $\mathsf{F}\ \phi$, that is we assume that ϕ is false in some model. A tableau proof represents a systematic search for such a model. Every tableau branch corresponds to a partial possible model in which the formulae on the branch are assigned the truth value corresponding to their sign. Therefore, a complementary pair of formulae, and thus a closed branch, denotes an explicit contradiction, since in every model each formula has a unique truth value.

Definition 2.6 (Closed Tableau, Tableau Provable) *A tableau is closed iff all of its branches are closed. An* **L***-formula ϕ is classically tableau provable, in symbols $\vdash_c \phi$, iff there exists a closed classical tableau for $\{\mathsf{F}\ \phi\}$.*

Definition 2.7 (Complete Branch, Complete Tableau) *A tableau branch is complete iff it is either closed or no rule application to a formula on the branch produces a formula that was not already present. A tableau is complete iff each of its branches is.*

A tableau proof tree represents a proof of the negated root formula when all branches in the tree can be closed simultaneously, in other words, when every attempt to construct a model that makes the root formula false leads to a contradiction.

At this point a few remarks are in order:

Remark 2.8 (Closure of Branches)

- *If* 0*-ary propositional connectives such as* **t** *(which evaluates constantly to* 1*) and* **f** *(which evaluates constantly to* 0*) are present additional closure conditions for branches become necessary. The constant operators* **t**, **f** *are handled by letting branches also be closed when they contain one of the formulae* $\mathsf{T}\,\mathbf{f}$,$\mathsf{F}\,\mathbf{t}$.
- *Thus the following alternate definition of branch closure, which will prove useful in the following, is motivated.*

  Alternate Definition of Branch Closure:

  Let $\mathbf{Contr}_c = \{\{\mathsf{T}\ \phi, \mathsf{F}\ \phi\} \mid \phi \in \mathbf{L}\} \cup \{\{\mathsf{T}\,\mathbf{f}\}, \{\mathsf{F}\,\mathbf{t}\}\}$ *be the contradiction set for classical tableaux. Then a tableau branch* **B** *is closed iff* $2^{\mathbf{B}} \cap \mathbf{Contr}_c \neq \emptyset$.
- *From now on we will use this definition. In all tableau systems for the various logics that we will consider, what will change besides the formula syntax are the tableau expansion rules and the choice of the contradiction set.*
- *Finally, it is sufficient to consider complementary* pairs *of formulae on the atomic level.*

Theorem 2.9 (Soundness, Completeness) *Let $\mathcal{L}$ be a classical first-order logic and let ϕ be any* **L***-sentence. Then there is a closed classical tableau for $\{\mathsf{F}\ \phi\}$ over $\mathbf{L}^*$ iff ϕ is a first-order classical tautology. In symbols*

$$\models_{\mathcal{L}} \phi \text{ iff } \vdash_c \phi.$$

Proof: See, for example, (Fitting, 1990). ∎

Remark 2.10 (Deletion of used formulae) *It is is sufficient for completeness to apply α-, β- and δ-rules only once to every formula in each branch. Consequently, formulae of these types may be deleted locally to the current branch after rule application. Note, however, that γ-formulae must be used repeatedly sometimes and hence may not be removed.*

Remark 2.11 (Systematic Tableaux and Fairness) *Tableau construction for a set of formulae Φ is a highly non-deterministic procedure. We did not specify, for example, in which order the tableau rules should be applied to the formulae on a branch, in which order newly generated branches should be processed, or what terms in which order should be "guessed" by the γ-rules. Somewhere on the way to an actual implementation, however, these questions have to be addressed, since any real program on a real machine behaves deterministically. Our completeness result, on the other hand, does not exhibit anything of the order in which the tableau is built up.*

If one wishes to extend the completeness result toward a concrete implementation the notion of systematic tableaux (Smullyan, 1968) *usually is introduced. The main issue to be paid attention to are the γ-rule applications. If the γ-rule is repeatedly applied to the same formula every term in the language finally must occur in the conclusion. Moreover, the order of rule applications to γ-formulae is to be "fair". Basically, this means that a tableau construction is carried out in such a way that to every γ-formula the corresponding rule is applied arbitrarily often. Depending on the rules that are used, fairness conditions can grow quite complex, in particular when other optimizations, like indexing of formulae (see Section 5.12.3) or free variables (see the following remark) are implemented.*

Fairness strategies in $_3T^AP$ are discussed in Sections 5.3.1.1, 5.4.2, 2.6.6.3, 2.6.6.4 and 5.11.4.

Remark 2.12 (Ground vs. Free Variable Tableaux) *One of the most important optimizations in tableau-based theorem proving deals with the γ-rules. Instead of guessing an arbitrary ground term, as it is done in the γ-rule we are using here, one does mark the quantified variable in the conclusion as* free *and it is instantiated only later with a term that is actually needed for a branch closure. Of course something has to be done then also with the δ-rules in order to preserve soundness. For obvious reasons we speak of a* free variable tableau system *when rules of this kind are used and of a* ground tableau system *when the present rules are used.*

In recent papers by Hähnle & Schmitt (Hähnle & Schmitt, 1993) *and Beckert, Hähnle & Schmitt* (Beckert *et al.*, 1993) *the earlier result of* (Fitting, 1990) *has been improved.*

We give the free quantifier rules for classical logic from (Hähnle & Schmitt, 1993) *in Table 2.3. The results carry over to the many-valued case without any restrictions or modifications.*

$\dfrac{\gamma}{\gamma(x)}$	$\dfrac{\delta}{\delta(f(x_1,\ldots,x_n))}$
where x is a free variable.	where $x_1,\ldots,x_n$ are the free variables occurring in δ and f is a new function symbol.

Table 2.3: Liberalized free tableau rules for quantified formulae.

The proviso of the δ-rule ensures that the introduced Skolem term preserves satisfiability of the current branch, even when the free variables become instantiated later during the proof.

In (Beckert *et al.*, 1993) *it has been shown that in two-valued logic the δ-rule can be even more liberalized: if the δ-rule is applied to formulae that are identical up to variable renaming, the same Skolem function symbol can be used multiply.*

Remark 2.13 (Strong Soundness and Completeness) *In classical logics, strong soundness as well as completeness can easily be proved by observing that for all classical logics $\mathcal{L}$ and all first-order sentences $\phi_1, \ldots, \phi_n, \psi$*

$$\{\phi_1, \ldots, \phi_n\} \models_{\mathcal{L}} \phi \text{ iff } \models_{\mathcal{L}} (\phi_1 \wedge \ldots \wedge \phi_n) \supset \phi$$

holds. This fact, which is a kind of deduction theorem, does not hold in most many-valued logics, and worse, consequence is not necessarily characterizable by finite logics at all. We will, therefore, in the following mostly be not concerned with consequences, but only with tautologies. For some many-valued logics, however, deduction theorems can be formulated.

We conclude this section with two small examples which have merely the purpose to introduce our notation for tableau proof trees.

Example 2.14 *With the tree drawn on the left we prove that*

$$\vdash_c (\exists x)(\forall y)r(x,y) \supset (\forall y)(\exists x)r(x,y) \ ,$$

while the tree on the right proves

$$\vdash_c p \supset (q \supset p) \ .$$

Formulae marked with an asterisk are being removed during the construction.

The formulae on the trees are numbered in the order of their appearance, starting with (1). *These numbers are enclosed by round brackets. The numbers in square brackets indicate the number of the parent formula. Beneath each closed branch the numbers of the formulae which led to the closure are given.*

$$\begin{array}{rl}
*\ (1)\ [-] & \mathsf{F}\,(\exists x)(\forall y)r(x,y) \supset (\forall y)(\exists x)r(x,y) \\
 & | \\
*\ (2)\ [1] & \mathsf{T}\,(\exists x)(\forall y)r(x,y) \\
 & | \\
*\ (3)\ [1] & \mathsf{F}\,(\forall y)(\exists x)r(x,y) \\
 & | \\
(4)\ [2] & \mathsf{T}\,(\forall y)r(c,y) \\
 & | \\
(5)\ [3] & \mathsf{F}\,(\exists x)r(x,d) \\
 & | \\
(6)\ [4] & \mathsf{T}\,r(c,d) \\
 & | \\
(7)\ [5] & \mathsf{F}\,r(c,d) \\
 & \textit{closed with } (6,7)
\end{array}
\qquad
\begin{array}{rl}
*\ (1)\ [-] & \mathsf{F}\,p \supset (q \supset p) \\
 & | \\
(2)\ [1] & \mathsf{T}\,p \\
 & | \\
*\ (3)\ [1] & \mathsf{F}\,q \supset p \\
 & | \\
(4)\ [3] & \mathsf{T}\,q \\
 & | \\
(5)\ [3] & \mathsf{F}\,p \\
 & \textit{closed with } (2,5)
\end{array}$$

On the left side, in (4) *a new Skolem constant c has been introduced via a δ-rule application to* (2), *while* (6) *was inferred from* (4) *by a γ-rule application, whereby y was instantiated with d. Similarly,* (7) *was inferred from* (5) *and* (5) *in turn from* (3). *The first rule applied to the tree was the α-rule on* (1), *corresponding to* F *and implication, thus generating formulae* (2) *and* (3).

The example on the right should be obvious enough.

For some more sophisticated examples of classical proof trees, see (Smullyan, 1968).

2.2 Dissolution

Dissolution is a sound and complete inference rule for classical first-order logic that has been introduced in 1986 by Murray & Rosenthal (Murray & Rosenthal, 1986; Murray & Rosenthal, 1987), see (Murray & Rosenthal, 1990a; Murray & Rosenthal, 1993) for a detailed description. It operates on formulae in prenex negation normal form that are built up from n-ary conjunctions and disjunctions. It has some similarity with Bibel's connection method (Bibel, 1987) in that the focus is on maximal conjunctive paths through the formula tree. It can also be seen as a refinement of a proof method by Prawitz (Prawitz, 1970).

In order to keep things simple, we constrain our exposition to the propositional case. Consider a formula built up from $\wedge, \vee$ and $\neg$ in NNF, which is represented as a graph, where nodes are formulae and edges are either of conjunctive or disjunctive type.

In the left part of Figure 2.1 we have drawn the graph for $\phi = D \wedge (A \vee B) \wedge (\overline{A} \vee C)$. Conjunctive connections are drawn vertically, while disjunctive connections are drawn horizontally.

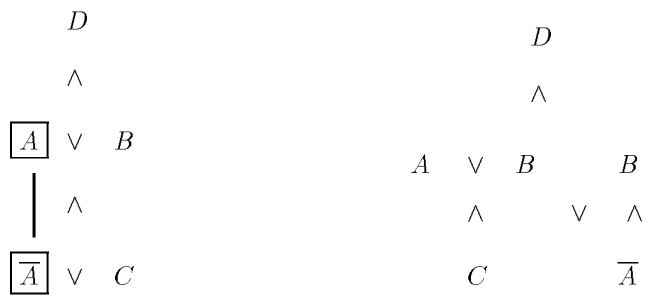

Figure 2.1: Dissolution for classical propositional logic.

Now consider maximal sets of conjunctively connected literals, so-called *c-paths*[2] in ϕ. For the present example, these are

$$\{\{D, A, \overline{A}\}, \{D, A, C\}, \{D, B, \overline{A}\}, \{D, B, C\}\}.$$

A pair of complementary literals lying on the same c-path is called a *link*. The dissolution rule operates always on a link in focus.[3] The central idea behind dissolution is to restructure a formula in such a way that exactly the c-paths containing the link in focus are removed. The right part in Figure 2.1 shows ϕ after dissolving on the link $(A, \overline{A})$, which is highlighted on the left hand side. One observes that the set of c-paths now is

$$\{\{D, A, C\}, \{D, B, \overline{A}\}, \{D, B, C\}\},$$

where the one path containing $(A, \overline{A})$ has been removed. The completeness of the method now follows from the fact that after a finite number of steps there can be no more c-paths left that contain any links. Since in an unsatisfiable formula each c-path must contain at least one link,

[2] We will not define formally the graph-based notions that are used in the following. The intuitive reading should be clear and we refer the reader to (Murray & Rosenthal, 1990a) for the exact definitions.

[3] In practice, that is. In general, a multiple-link dissolution rule can be defined. The multiple-link rule represents also an alternative approach to many-valued dissolution. It turns out, however, that implementation and control of multiple-link rules is not feasible in practice and hence will not be considered here.

the empty graph must be produced after a finite number of dissolution steps if and only if the starting formula was unsatisfiable.

In (Murray & Rosenthal, 1990b) it was shown that already in the propositional case the restricted application of dissolution to tableau situations (that is, the input formulae containing the link must be on the same branch) can yield substantially shorter tableau proofs.

The dissolution rule built into $_3T^AP$ does also function for first-order formulae. In that case the dissolution step involves unification of the complementary link literals and, as a consequence, possible instantiations of free variables.

The version of dissolution for tableaux which is used in $_3T^AP$ is described in Section 5.11. For a deepened discussion of dissolution and tableaux, see (Murray & Rosenthal, 1990b; Kreidler, 1992).

2.3 Many-Valued Logic

The formal language of many-valued logic is essentially the same as that for classical logic. The only difference is that now and then some additional unary or binary propositional connectives will occur.

Very much as in classical logic, we can specify the semantics of a many-valued logic with either truth tables or by recursive definition of a meaning function $v_{\mathbf{M}}$ which assigns for every model $\mathbf{M}$ to each formula a truth value. The only difference is that the set of truth values does no longer consist of only two elements, but any finite number.[4] We will take for the set of truth values N natural numbers, that is $N = \{0, 1, 2, \ldots, n-1\}$ for an n-valued logic.

As an example take three-valued conjunction. The truth table

$\wedge$	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

and

$$v_{\mathbf{M}}(\phi \wedge \psi) := \min\{v_{\mathbf{M}}(\phi), v_{\mathbf{M}}(\psi)\} \text{ where } 0 < 1 < 2$$

define exactly the same three-valued connective.

A subset D of the set of truth values will be called designated. The truth values in D play the role of *true* in the classical case and support validity.

A model is simply a mapping from atomic formulae into N, in other words the determination of $v_{\mathbf{M}}$ on atoms which is uniquely extended to all formulae. A formula ϕ is satisfiable iff $v_{\mathbf{M}}(\phi) \in D$ for some $\mathbf{M}$. If a formula ϕ is satisfied by any model it is called tautology.

We sketch a somewhat naïve method of extending tableaux to handle any finitely-valued first-order logic. The method is due to S. Surma who presented it on the International Symposium on Multiple-Valued Logic in 1974 (Surma, 1984). Some years later, W. Carnielli (Carnielli, 1987) filled the gaps in Surma's somewhat sketchy presentation and extended the treatment to a broad class of many-valued quantifiers, including the standard ones that we consider.

[4] Or even an infinite number of truth values. $_3T^AP$ can handle only finitely many truth values, however.

Assume for the moment that we are working in a three-valued logic. Then, obviously, saying that a formula ϕ is not true is not equivalent to saying that it is false, or more precisely saying that ϕ has not truth value 2 is not equivalent to saying that it has truth value 0. Yet another formulation of this fact, with respect to signed tableaux, is that *not* $\mathsf{T}\ \phi$ is not the same as $\mathsf{F}\ \phi$. But being able to express this fact, that *not* $\mathsf{T}\ \phi$, is crucial for the tableau method to work, since this is what we must put in the initial tableau if the tautologyhood of ϕ is to be established.

The solution is to introduce other signs than T and F , namely as many as there are truth values in a logic. Let us fix

$$\mathbf{S}_{\mathcal{L}} = \{\mathsf{0}, \mathsf{1}, \ldots, \mathsf{n} - \mathsf{1}\}$$

as the set of signs for an n-valued logic in this section. Each sign corresponds to a truth value in an obvious way. We use the same symbols for signs and truth values, only the former are printed in Sans Serif typeface and understand this convention as an implicit type conversion function. We are now able to express invalidity of a formula in the following way:

$$\textit{not}\ 2\ \phi \text{ iff } \mathsf{0}\ \phi \text{ or } 1\ \phi \tag{2.1}$$

So far, so good—but how does one compute the tableau rules corresponding to a certain sign and connective in this new setting? Assume that we wanted to compute the rule corresponding to 1 and conjunction for $n = 3$. If we take a look at the truth table above we see that there are three entries that are equal to 1 (which corresponds to 1). From these entries we can extract the necessary and sufficient conditions on the direct subformulae of a formula $\phi \wedge \psi$ that characterize the assertion $1\ (\phi \wedge \psi)$. More precisely, we have that

$$1\ (\phi \wedge \psi) \quad \text{iff} \quad (1\ \phi \text{ and } 2\ \psi) \text{ or } (1\ \phi \text{ and } 1\ \psi) \text{ or } (2\ \phi \text{ and } 1\ \psi).$$

Transforming this into a tableau rule, we get

$$\begin{array}{c|c|c} \multicolumn{3}{c}{1\ (\phi \wedge \psi)} \\ \hline 1\ \phi & 1\ \phi & 2\ \phi \\ 2\ \psi & 1\ \psi & 1\ \psi \end{array}$$

With the same method one can compute rules for all combinations of signs and connectives, provided the truth tables are known and the sign *does occur* in the truth table of the connective. If it does not the current branch can be closed at once, since an assertion of that kind can never be satisfied. A convenient method to handle these cases is to include them in the contradiction set of a tableau system.

Let us collect some immediate observations:

1. The rules in general fall neither into the α- nor into the β-schema.

2. The number of extensions generated by a rule for a formula $\mathsf{S}\phi$ can be equal to the number of entries corresponding to S in the truth table of the leading connective of ϕ. For n truth values and k-ary connectives the worst case is a branching factor of $n^k - n$ with $k(n^k - n)$ formulae in the conclusion. Since every entry in the truth table has to be analyzed in exactly one of the rules corresponding to the connective, the number of extensions in all rules for a connectives is but in rare cases, when simplifications are possible, equal to n^k.

$\supset$	0	1	2
0	2	2	2
1	2	2	2
2	0	1	2

	$\neg$	$\sim$
0	2	2
1	1	2
2	0	0

Table 2.4: Truth tables of propositional connectives for $n = 3$.

3. The rules themselves are uniquely determined up to the ordering of the extensions in the conclusion and the formulae within each extension.
4. With equation (2.1) we can express the assertion that a formula is not satisfiable. The price, however, is that the construction of *two* tableaux is required. In general, when $n > 3$, as many tableaux as there are non-designated truth values are required.

Example 2.15 *In Figure 2.2 the proof trees corresponding to the proof of the three-valued tautology $\neg p \supset (\sim p \wedge \neg p)$ are shown. The truth tables of $\sim$, $\neg$ and $\supset$ are given in Table 2.4. Note that we need two proof trees in order to refute the two non-designated truth values* 0 *and* 1 *. Formulae in the left tree correspond to formulae with same numbers in the right tree.*

For first-order logic we define many-valued generalizations of the usual quantifiers $\exists$ and $\forall$ by treating them as infinitary disjunctions and conjunctions, resp.:

1. $v_\beta((\forall y)\phi) = \min\{v_{\beta_y^u}(\phi) | u \in U\}$, where min is interpreted naturally on N.
2. $v_\beta((\exists y)\phi) = \max\{v_{\beta_y^u}(\phi) | u \in U\}$, where max is interpreted naturally on N.

β is a variable assignment and β_y^u is the same as β, but with the value of y changed to u. The notation is completely analogous to classical logic, cf. (Fitting, 1990).

First-order logic is handled as well by Carnielli's approach, but the resulting tableau rules are rather bulky. We do not give them here.

2.4 Lemma Generation

Consider the application of a (symmetric) β-rule to a formula on the current branch, say to F $(\phi \wedge \psi)$. One has to make a decision which branch is put into focus for the next rule application. This decision generally is based on some heuristics. We are not interested right now on which, but it is clear that a decision has to be made somehow. So one of the newly generated branches will be closed first, say the left one. Obviously it would have been better then to use an asymmetric rule, namely the one in the middle in Table 2.5 which gives us more information about the still open branch. And this is exactly what we do: Before actually applying a β-rule, decide which branch is put into focus next. Then apply an asymmetric version of the rule which provides more information on the branch processed last.

We call this technique *Lemma Generation*, because if we interpret the additional formula in an asymmetric rule as a lemma the branch that is closed first can be seen as a proof for that lemma. Lemma generation is not merely an *ad hoc* efficiency hack, rather it lifts classical analytic tableaux to a *better proof length complexity class.*

In $_3T^AP$ lemma generation can be switched on with two different priorities. It works also in many-valued logics, but then it is not easy to compute adequate lemmata. See (Hähnle, 1992c, p. 103ff) for the details. Technically, lemma generation can be achieved by changing the rule set and the heuristics for rule selection, see Section 5.8.2.

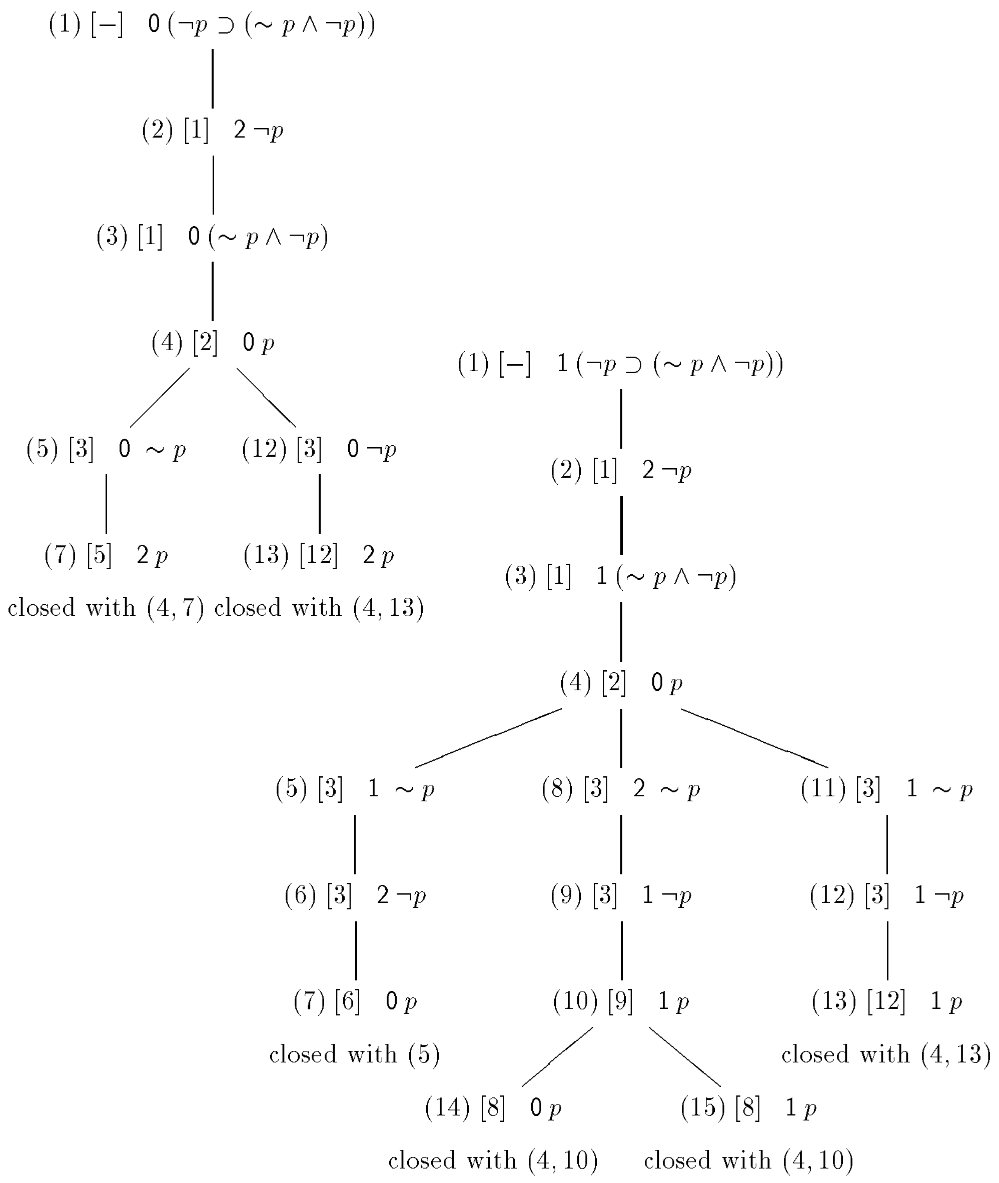

Figure 2.2: Tableau proof of $\neg p \supset (\sim p \wedge \neg p)$ using the method of Surma and Carnielli.

$$\begin{array}{c} \mathsf{F}\,(\phi \wedge \psi) \\ \hline \mathsf{F}\,\phi \mid \mathsf{F}\,\psi \end{array} \qquad \begin{array}{c|c} \multicolumn{2}{c}{\mathsf{F}\,(\phi \wedge \psi)} \\ \hline \mathsf{F}\,\phi & \mathsf{F}\,\psi \\ & \mathsf{T}\,\phi \end{array} \qquad \begin{array}{c|c} \multicolumn{2}{c}{\mathsf{F}\,(\phi \wedge \psi)} \\ \hline \mathsf{F}\,\phi & \mathsf{F}\,\psi \\ \mathsf{T}\,\psi & \end{array}$$

Table 2.5: Symmetric and asymmetric tableau rules.

Note that in first-order logics lemma generation is not always advantageous, since it may enlarge the search space.

2.5 Universal Formulae

Formulae, and in particular equalities, have often to be applied more than once in order to close a branch, each time with different substitutions for the free variables occurring in them. A typical example is the associativity axiom $As = (\forall x)(\forall y)(\forall z)[(x \cdot y) \cdot z \approx x \cdot (y \cdot z)]$ from group theory. In most cases it has to be applied several times with different instantiations of x, y and z to prove even simple theorems of group theory.

In semantic tableaux the mechanism to do so usually is to apply the γ-rule more than once to As and thus generate several instances of As, each with different free variables substituted for x, y and z.

Consequently, to prove a theorem the γ-limit q has to be at least as high as the maximal number of necessary applications of the same formula with different substitutions for the free variables it contains. However, the higher the limit q is, the more branches have to be closed and the bigger the tableau becomes. Moreover, it is quite difficult to choose the limit q appropriately, because one does not know how many instances of γ-formulae will be needed.

For equalities, the problem could be avoided if they were not allowed to occur nested in other formulae, but appeared only on the top-level. We did, however, not want to employ this restriction in order to allow for a natural formulation of problems. Nevertheless, the problem can at least partly be solved if one is able to recognize formulae (and in particular equalities) whose universal closure is strongly implied by the formulae on the branch. These are "locally universal"; they can be used repeatedly with different substitutions for the variables they contain.

Definition 2.16 (Strong Consequence Relation) *Let ϕ, ψ be first-order formulae;*

$$\phi \models^{\circ} \psi$$

if for all interpretations I and for all variable assignments β:

$$\textit{if } \mathrm{val}_{I,\beta}(\phi) = \textit{true} \;\; \textit{then } \mathrm{val}_{I,\beta}(\psi) = \textit{true}$$

Remark 2.17 *Note, that for example $p(x) \not\models^{\circ} (\forall x p(x))$, but $p(x) \models (\forall x p(x))$, where "$\models$" denotes the weak consequence relation.*

Definition 2.18 (Universal Formula) *Suppose ϕ is a formula on some tableau branch B. ϕ is universal on B with respect to the variable x if*

$$B \models^{\circ} (\forall x\, \phi)\ .$$[5]

Now, we can use a new rule for closing branches that takes this definition into account:

Definition 2.19 (Closed Tableau with Universal Formulae) *A tableau consisting of the k branches $B_1, \ldots, B_k$ is closed if there are*

1. *a substitution σ,*
2. *literals l_i, $\bar{l}_i \in B_i$, and*

[5] In the sequel, we will often refer to a formula ϕ which is universal on a branch B w.r.t. a variable x just by "the universal formula ϕ", and to the variable x by "the universal variable x" (if the context is clear).

3. *substitutions σ_i, such that*
   (a) *$l_i\sigma_i$ and $\bar{l}_i\sigma_i$ are complementary, and*
   (b) *if $\sigma_i(x) \neq \sigma(x)$ then both l_i and $\bar{l}_i$ are universal on B_i w.r.t. x.*

With this definition of closed tableaux it is possible that a tableau is closed after less applications of expansion rules than in the standard free-variable tableau calculus. Thus, the calculus is strengthened.

The problem of recognizing universal formulae is of course undecidable in general. However, a wide and important class can be recognized quite easily: assume there is a sequence of tableau rule applications that does not contain a disjunctive rule (i.e., the tableau does not branch). All formulae that are generated by this sequence are universal w.r.t. the free variables introduced by the sequence. Substitutions for these variables can be ignored, since the corresponding inference steps could be repeated arbitrarily often to generate new instances of the universal variables (without generating new branches).

More formally, we use the following theorem:

Theorem 2.20 *A formula ϕ on a branch B is universal w.r.t. x if ϕ was put on B by either*

1. *applying a γ-rule and x is the free variable introduced by the application of this rule, or*
2. *applications of non-branching rules to a formula $\psi \in B$, where ψ is universal on B w.r.t. x.*

Recognizing the above subset of universal formulae is implemented in $_3T^AP$ by keeping a list of in this sense universal variables for each formula.

2.6 Equality Handling

2.6.1 Introduction

One of the main goals of Automated Deduction is to efficiently handle first-order logic *with equality.* In this section we describe how "mixed" E-unification (Beckert, 1994b), a combination of the classical "universal" E-unification and "rigid" E-unification (Gallier *et al.*, 1992), can be used to efficiently handle equality in free variable and in universal formula semantic tableaux (see Section 2.5).

A more detailed description of this new approach can be found in (Beckert, 1993b), including proofs for its soundness and completeness, examples, and an analysis of the shortcomings in previous approaches (the important results, but without proofs, can be found in (Beckert, 1994b; Beckert, 1994a), too).

2.6.2 Syntax and Semantics of Equality

Let us assume that the set **R** of predicate symbols of the first-order language **L** contains a binary predicate symbol for equality which we denote by $\approx$ such that no confusion with the meta-level equality predicate $=$ can arise. We stress that there is no restriction where equalities can occur in formulae.

We use sequences of natural numbers to denote positions in terms; t_p is the subterm at position p in the term t (e.g. $f(a,b)_{\langle 2\rangle} = b$).

For the sake of simplicity and without any loss of generality, we use a slightly non-standard notion of substitutions: They have to be idempotent and of finite domain; **Subst** is the set of these substitutions. *id* is the empty substitution. The application of a substitution σ to a term t is denoted by $t\sigma$; if a substitution is applied to a quantified rule, equality, or term, the bound variables are never instantiated. $\leq$ denotes the specialization relation on substitutions: $\sigma \leq \tau$ iff there is a σ' such that $\sigma' \circ \sigma = \tau$.

A model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ (with domain $\mathbf{D}$ and interpretation $\mathbf{I}$) is called **normal** iff $\approx^{\mathbf{I}}$ is the identity relation on $\mathbf{D}$. A model is called **canonical** if, moreover, for every $d \in \mathbf{D}$ there is a term t in $\mathbf{L}$ such that $t^{\mathbf{I}} = d$. The following theorem shows that canonical models are analogous to Herbrand models.

Theorem 2.21 *If a set S of universal sentences is satisfied by a normal model, then there is also a canonical model that satisfies S.*

2.6.3 Equality Handling by Mixed E-Unification

Constructing a tableau for a first-order formula ϕ can be considered a search for a model of ϕ. Therefore, as part of the tableau calculus, methods have to be employed for: (i) adding formulae that are valid in a model $\mathbf{M}$ of ϕ to the tableau branch that corresponds to $\mathbf{M}$ (i.e., that is a partial definition of $\mathbf{M}$), and (ii) recognizing formulae or sets of formulae that are unsatisfiable; these formulae close branches on which they occur.

In *canonical* models, on the one hand, additional formulae are valid and, thus, have to be added to a branch: If $P(a)$ and $a \approx b$ are true in a canonical model $\mathbf{M}$, then $\mathbf{M}$ is a model of $P(b)$, too. On the other hand, there are additional inconsistencies: $\neg(a \approx a)$ is false in all *canonical* models.

Accordingly, there are two techniques for handling equality in semantic tableaux: The first and more straightforward method is to define additional tableau rules for expanding branches by all the formulae valid in the canonical models they (partially) define; then very simple additional closure rules can be used (Jeffrey, 1967; Reeves, 1987; Fitting, 1990). The second possibility is to use a more complicated notion of closed tableaux: *E-unification* is used to decide whether a tableau branch is unsatisfiable in canonical models and, therefore, closed. Then, no additional expansion rules are needed.

The common problem of all the methods for handling equality, that are based on additional tableau expansion rules, is that there are virtually no restrictions on the "application" of equalities. This leads to a very large search space; even very simple problems cannot be solved in reasonable time.

It is difficult to employ more elaborate and efficient methods for handling equality in semantic tableaux, such as completion-based approaches, because it is nearly impossible to transform these methods into (sufficiently) simple tableau expansion rules. Contrary to that, arbitrary algorithms can be used, if the handling of equality is reduced to solving E-unification problems.

2.6.4 Universal, Rigid and Mixed E-Unification

The intention of defining different versions of E-unification is to allow equalities to be used differently in a proof: in the universal case the equalities can be "applied" several times with different instantiations for the variables they contain; in the rigid case they can be "applied" more than once but with only one instantiation for each variable they contain; in the mixed case there are both types of variables. To distinguish the different types of variables syntactically, equalities can be explicitly quantified:

Definition 2.22 *A* mixed E-unification problem

$$\langle E, s, t\rangle$$

consists of a finite set E of equalities of the form $(\forall x_1)\cdots(\forall x_n)(l \approx r)$ and terms s and t.[6]
A substitution σ is a solution *to the problem, iff*

$$E\sigma \models (s\sigma \approx t\sigma) \ ,$$

where the free variables in $E\sigma$ are "held rigid", i.e. treated as constants.[7]
A mixed E-unification problem $\langle E, s, t\rangle$ is called purely universal *if there are no free variables in E, and* purely rigid *if there are no bound variables in E.*

The major differences between this definition and that generally given in the (extensive) literature on (universal) E-unification are:

1. The equalities in E are *explicitly* quantified (instead of considering all the variables in E to be *implicitly* universally quantified).
2. In difference to the "normal" notion of logical consequence, free variables in $E\sigma$ are "held rigid".
3. The substitution σ is applied not only to the terms s und t but as well to the set E.

Example 2.23 *All substitutions are solutions to the purely universal problem*

$$\langle\{(\forall x)(f(x) \approx x)\},\ g(f(a), f(b)),\ g(a, b)\rangle \ .$$

The (very similar) purely rigid problem

$$\langle\{(f(x) \approx x)\},\ g(f(a), f(b)),\ g(a, b)\rangle$$

has no solution.
$\{y/b\}$ is a solution to the mixed problem

$$\langle\{(\forall x)(f(x, y) \approx f(y, x))\},\ f(a, b),\ f(b, a)\rangle \ ;$$

since the variable x is quantified, it does not have to be instantiated by the unifier.

For handling equality in semantic tableaux, several E-unification problems have to be solved simultaneously (one for each branch):

Definition 2.24 *A finite set*

$$\{\langle E_1, s_1, t_1\rangle, \ldots, \langle E_n, s_n, t_n\rangle\} \qquad (n \geq 1)$$

of mixed E-unification problems is called simultaneous *E-unification problem.*
A substitution σ is a solution to the simultaneous problem iff it is a solution to every component $\langle E_k, s_k, t_k\rangle$ $(1 \leq k \leq n)$.

Since purely universal E-unification is already undecidable, (simultaneous) *mixed* E-unification is—in general—undecidable as well. Is is, however, possible to enumerate a complete set of most general unifiers. (Simultaneous) *purely rigid* E-unification is decidable (Gallier *et al.*, 1992; Goubault, 1993).[8]

[6] Without making a real restriction, we require the sets of bound and free variables in the problem to be disjoint.
[7] This is equivalent to $E\sigma \models^{\underline{\circ}} (s\sigma \approx t\sigma)$.
[8] Purely rigid E-unification is NP-complete (Gallier *et al.*, 1992); simultaneous purely rigid E-unification is NEXPTIME-complete (Goubault, 1993).

2.6.5 Extracting E-Unification Problems from Tableaux

The equality theory defined by a tableau branch B consists of the equalities on B; they are (explicitly) quantified w.r.t. to the variables w.r.t. which they can be recognized as being universal:

Definition 2.25 *Let B be a tableau branch. The* set $E(B)$ of equalities *consists of the equalities*

$$(\forall x_1) \cdots (\forall x_n)(s \approx t)$$

such that

1. $\mathsf{T}\,(s \approx t)$ *is formula on B,*
2. $\mathsf{T}\,(s \approx t)$ *is recognized as being universal w.r.t. $\{x_1, \ldots, x_n\}$ on B.*[9]

There are unification problems for each inequality on a branch B and each pair of atoms that potentially close B, i.e., atoms with the same predicate sign and complementary truth value signs:

Definition 2.26 *Let B be a tableau branch. The* set $\mathcal{P}(B)$ of unification problems *consists exactly of the sets of term pairs:*

$$\{\langle s_1\nu, t_1\nu\rangle, \ldots, \langle s_n\nu, t_n\nu\rangle\}$$

for each pair

$$\mathsf{T}\,P(s_1, \ldots, s_n), \mathsf{F}\,P(t_1, \ldots, t_n) \in B$$

of (potentially closing) atoms such that $P \neq \approx$, and

$$\{\langle s\nu, t\nu\rangle\}$$

for each inequality

$$\mathsf{F}\,(s \approx t) \in B \ .$$

The substitution $\nu = \{x_1/y_1, \ldots, x_m/y_m\}$ renames all the variables $x_1, \ldots, x_m$ w.r.t. which both $\mathsf{T}\,P(s_1, \ldots, s_n)$ and $\mathsf{F}\,P(t_1, \ldots, t_n)$ are recognized as being universal (resp. w.r.t. which $\mathsf{F}\,(s \approx t)$ is recognized as being universal); $y_1, \ldots, y_m$ are new variables.

If one of the problems in the set $\mathcal{P}(B)$ of unification problems of a branch B has a solution σ (w.r.t. the equalities $E(B)$), $B\sigma$ is unsatisfiable in canonical models; therefore the branch B is closed under the substitution σ. The pair of potentially closing atoms corresponding to the solved unification problem has been proven to actually be complementary; or the corresponding inequality has been proven to be inconsistent (provided the unifier is applied to the tableau).

The following is a formal definition of the simultaneous mixed E-unification problems that have to be solved to close a tableau:

Definition 2.27 *A universal formula tableau T with branches $B_1, \ldots, B_k$ is* closed *iff in the sets of unification problems $\mathcal{P}(B_i)$ there are elements*

$$\{\langle s_{i1}, t_{i1}\rangle, \ldots, \langle s_{in_i}, t_{in_i}\rangle\} \in \mathcal{P}(B_i)$$

$(1 \leq i \leq k)$ such that there is a solution to the simultaneous mixed E-unification problem

$$\begin{array}{l} \{\ \langle E(B_1), s_{11}, t_{11}\rangle,\ \ldots,\ \langle E(B_1), s_{1n_1}, t_{1n_1}\rangle, \\ \quad\ \vdots \qquad\qquad\quad \vdots \qquad\qquad \vdots \\ \ \ \langle E(B_k), s_{k1}, t_{k1}\rangle,\ \ldots,\ \langle E(B_k), s_{kn_k}, t_{kn_k}\rangle\ \} \end{array}$$

(see Definitions 2.25 and 2.26).

[9] An arbitrary method for recognizing universal formulae may be used; $_3T^AP$ uses the method from Theorem 2.20.

Actually, it is not necessary to split pairs

$$\mathsf{T}\,P(s_1,\ldots,s_n) \text{ and } \mathsf{F}\,P(t_1,\ldots,t_n)$$

of potentially complementary atoms into n term pairs

$$\langle s_1,t_1\rangle,\ldots,\langle s_n,t_n\rangle$$

that have to be unified. Instead the single problem

$$\langle P(s_1,\ldots,s_n), P(t_1,\ldots,t_n)\rangle$$

could be used. That, however, is inefficient, because the n simpler problems can be solved independently.

Example 2.28 *As an example we use the tableau from Figure 2.3. Its left branch is denoted by B_1 and its right branch by B_2. If the method for recognizing universal formulae from Theorem 2.20 is used, $E(B_2)$ contains the equalities*

$$b \approx c \text{ and } (\forall x)(g(f(x)) \approx x) \enspace .$$

$E(B_1)$ contains in addition the equality

$$g(x_2) \approx f(x_2) \enspace .$$

Both $\mathcal{P}(B_1)$ and $\mathcal{P}(B_2)$ contain the set

$$\{\langle g(g(a)), a\rangle,\ \langle b, c\rangle\} \enspace .$$

$\mathcal{P}(B_2)$ contains in addition the set

$$\{\langle x_2, a\rangle\} \enspace .$$

The tableau is closed (Definition 2.27), because the substitution $\sigma = \{x_2/a\}$ is a solution to the simultaneous mixed E-unification problem

$$\begin{array}{l}\{\langle E(B_1),\ g(g(a)),\ a\rangle,\\ \ \langle E(B_1),\ b, \qquad\quad c\rangle,\\ \ \langle E(B_2),\ x_2, \qquad\ \ a\rangle\} \enspace .\end{array}$$

2.6.6 Solving Mixed E-Unification Problems

The Unfailing Knuth-Bendix-Algorithm (UKBA) (Knuth & Bendix, 1970; Bachmair *et al.*, 1989) with narrowing (Nutt *et al.*, 1989), that is generally considered to be the best algorithm for universal E-unification and has often been implemented, cannot be used to solve rigid or mixed problems. Completion-based methods for rigid E-unification have been described in (Gallier *et al.*, 1992; Goubault, 1993). These, however, are non-deterministic and unsuited for implementation (they have, in fact, never been implemented). In (Beckert & Hähnle, 1992) a method for solving mixed E-unification problems has been introduced that does not use completion but is based on computing equivalence classes.[10]

The basic idea of our approach—and the main difference to the classical unfailing completion procedure—is that during the completion process *free* variables are never renamed, even if equalities that have variables in common are applied to each other. In addition, constraints consisting of a substitution and an *order condition* are attached to the reduction rules and terms.[11]

[10] This method has been used in all earlier versions of $_3T^AP$, but it is now substituted by the method described in the following.

[11] In (Chabin *et al.*, 1993) a similar type of constraints is used for E-unification—but only for its purely universal version. In (Beckert & Hähnle, 1992) substitutions are used to restrict the validity of terms. For a completion-based approach, however, this is not sufficient because the validity of reduction rules depends on the ordering on terms.

$$\begin{tikzcd}
 & (1)\ \mathsf{T}\,(\forall x)((g(x) \approx f(x)) \lor \neg(x \approx a)) \arrow[d, dash] & \\
 & (2)\ \mathsf{T}\,(\forall x)(g(f(x)) \approx x) \arrow[d, dash] & \\
 & (3)\ \mathsf{T}\,(b \approx c) \arrow[d, dash] & \\
 & (4)\ \mathsf{T}\,P(g(g(a)), b) \arrow[d, dash] & \\
 & (5)\ \mathsf{T}\,\neg P(a, c) \arrow[d, dash] & \\
 & (6)\ \mathsf{F}\,P(a, c) \arrow[d, dash] & \\
 & (7)\ \mathsf{T}\,(g(f(x_1)) \approx x_1) \arrow[d, dash] & \\
 & (8)\ \mathsf{T}\,((g(x_2) \approx f(x_2)) \lor \neg(x_2 \approx a)) \arrow[dl, dash] \arrow[dr, dash] & \\
(9)\ \mathsf{T}\,(g(x_2) \approx f(x_2)) & & (10)\ \mathsf{T}\,\neg(x_2 \approx a) \arrow[d, dash] \\
 & & (11)\ \mathsf{F}\,(x_2 \approx a)
\end{tikzcd}$$

Figure 2.3: A free variable tableau for the given formulae (1) to (5). By applying the standard free variable tableau rules, formula (6) is derived from (5), (7) from (2), (8) from (1), (9) and (10) from (8), and (11) from (10). Formula (7) is recognized as being universal w.r.t. x by the method from Theorem 2.20.

2.6.6.1 Constraints

For different substitutions σ, the completion of $E\sigma$ contains different reduction rules. Nevertheless, a single completion can be computed for all $E\sigma$, if constraints are attached to the rules to restrict their validity to certain (sets of) substitutions.

The first part of the constraints we attach to reduction rules and terms is an *order condition*; it expresses a restriction on the ordering of terms w.r.t. the reduction ordering $\succ_{\text{LPO}}$.

The reduction ordering $\succ_{\text{LPO}}$ on terms is an arbitrary but fixed *lexicographic path ordering* (LPO) (Dershowitz, 1987) that is total on ground terms.

Definition 2.29 (Lexicographic Path Ordering) *A total ordering $>_F$ on the set of function symbols induces a lexicographic path ordering on terms: $s \succ_{\text{LPO}} t$, where $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$, iff*

1. *$s_i \succ_{\text{LPO}} t$ or $s_i = t$ for some $1 \leq i \leq m$, or*
2. *$f >_F g$ and $s \succ_{\text{LPO}} t_j$ for all $1 \leq j \leq n$, or*
3. *$f = g$, $(s_1, \ldots, s_n) \succ\!\!\succ_{\text{LPO}} (t_n, \ldots, t_n)$ and $s \succ_{\text{LPO}} t_i$ for all $1 \leq i \leq n$.*

$\succ\!\!\succ_{\text{LPO}}$ is the lexicographic ordering on term tupels induced by $\succ_{\text{LPO}}$, i.e.,

$$\langle s_1, \ldots, s_n \rangle \succ\!\!\succ_{\text{LPO}} \langle t_1, \ldots, t_n \rangle$$

there is an $1 \leq i \leq n$, such that $s_j = t_j$ for all $1 \leq j < i$ and $s_i \succ_{\text{LPO}} t_i$.

Example 2.30 *A reduction system equivalent to $E\sigma = \{x \approx y\}\sigma$ either consists of the rule $(x \rightarrow y)\sigma$ or the rule $(y \rightarrow x)\sigma$, depending on which of the terms $\sigma(x)$ and $\sigma(y)$ is greater w.r.t. the LPO used.*

The expression $x \succ y$ is the natural choice for a restriction such as "the term substituted for x has to be greater than that substituted for y":

Definition 2.31 Order conditions *are composed of the* atomic *order conditions $s \succ t$ (s and t are terms) using the logical connectives $\neg$, $\wedge$, $\vee$ and $\supset$, and the constants* true *and* false*.*

Ground order conditions, i.e., order conditions that contain no variables, are assigned a truth value by interpreting the (predicate) symbol $\succ$ by a (fixed) LPO.

A (non-ground) order condition O is true *iff $O\sigma$ is true for all ground substitutions σ,* false *(or inconsistent) iff its negation $\neg O$ is true, and* consistent *iff it is not false.*

Since LPOs are total on ground terms, the truth value of ground order conditions is well defined; non-ground order conditions are (similar to first order formulae) either consistent or inconsistent, and may be true or false.

Example 2.32 *The order condition $f(a) \succ a$ is true; $(x \succ y) \wedge (y \succ x)$ is false; and $x \succ y$ is consistent. The truth value of $a \succ b$ depends on the LPO used to interpret $\succ$.*

In some cases, order conditions are not sufficient for describing the set of substitutions for which a reduction rule is valid:

Example 2.33 *Suppose $E = \{f(b) \approx a, f(x) \approx c\}$; the reduction rule $c \rightarrow a$ is part of the completion of $E\sigma$ iff $\sigma(x) = b$ (then the equalities are a critical pair).*

One could use the formula $x \approx t$ to express conditions of the form "x has to be substituted by (an instance of) t", if the predicate symbol $\approx$ were allowed in order conditions. That, however, would make the handling of conditions unnecessarily complicated. Instead the substitution $\{x/t\}$ itself becomes part of the constraint:

Definition 2.34 *A* constraint $c = \langle \sigma, O \rangle$ *consists of a substitution σ and an order condition O such that the variables in the domain of σ do not occur in O, i.e. $O = O\sigma$.*

A substitution τ satisfies *a constraint $c = \langle \sigma, O \rangle$ iff τ is a specialization of σ and $O\tau$ is true. τ satisfies a set C of constraints iff there is a $c \in C$ satisfied by τ.*

$\mathrm{Sat}(c)$ *(resp.* $\mathrm{Sat}(C)$*) is the set of substitutions satisfying the constraint c (resp. the set C of constraints).*

Note, that sets of constraints implicitly represent *disjunctions*. To simplify the handling of constraints, we give some additional definitions and notations:

Definition 2.35 *A constraint c_1* subsumes *a constraint c_2 iff the substitutions satisfying c_2 satisfy as well c_1:* $\mathrm{Sat}(c_2) \subset \mathrm{Sat}(c_1)$.

A constraint c^{-1} that is satisfied by the substitutions not *satisfying c is called* negation *of c:* $\mathrm{Sat}(c^{-1}) = \mathbf{Subst} \setminus \mathrm{Sat}(c)$.

A constraint $c_1 \sqcap c_2$ that is satisfied by the constraints satisfying both c_1 and c_2 is called a combination *of c_1 and c_2:* $\mathrm{Sat}(c_1 \sqcap c_2) = \mathrm{Sat}(c_1) \cap \mathrm{Sat}(c_2)$.

The *empty constraint* $\epsilon = \langle id, true \rangle$ consists of the empty substitution id and the order condition $true$; it is satisfied by all substitutions.

There are efficient algorithms for computing negations and combinations of constraints. Since a constraint c_1 subsumes a constraint c_2 iff $c_1^{-1} \sqcap c_2$ is inconsistent, these are an important part of the implementation. Deciding whether a constraint is satisfiable is NP-hard (Comon, 1990). The problem can however simplified considerably: The order condition $(s \succ x) \wedge (x \succ t)$ is inconsistent if there is no term between s and t (w.r.t. the LPO used). Without causing any harm, we can do without checking for such inconsistencies, that are very difficult to detect (see Section 5.13.10).

2.6.6.2 Constrained Terms and Reduction Rules

Since—syntactically—constrained reduction rules can be considered to be constrained terms,[12] it suffices to define the latter:

Definition 2.36 *A* constrained term $\mathbf{t} = (\forall \bar{x})(t \ll c)$ *is a term t with a constraint $c = \langle \sigma, O \rangle$ attached to it such that $t\sigma = t$.*[13] *It can be universally quantified w.r.t. some or all of the variables it contains (the quantification includes the constraint).*

On first sight quantified terms may look strange, but, later on, a constrained term $\mathbf{t}$ is used to express the fact that it can be derived from another term $\mathbf{t}'$. Therefore, it is important to be able to make a distinction between rigid and non-rigid (quantified) variables.

Using constraints, for every equality an equivalent set of reduction rules can be constructed; even for those that cannot be oriented without constraints.

[12] Over a different signature that contains $\rightarrow$ as a function symbol.

[13] The symbol $\ll$ means "if".

Example 2.37 *The equality $f(x) \approx g(y)$ cannot be oriented without constraints, since (i) its instance $f(g(a)) \approx g(a)$ has to be oriented from left to right, while (ii) its instance $f(a) \approx g(f(a))$ has to be oriented from right to left. The* constrained *rules $f(x) \to g(y) \ll \langle id, f(x) \succ g(y)\rangle$ and $g(y) \to f(x) \ll \langle id, g(y) \succ f(x)\rangle$, however, define the same derivability relation as the equality $f(x) \approx g(y)$.*

Other typical examples are the constrained rules $x \to y \ll \langle id, x \succ y\rangle$ and $y \to x \ll \langle id, y \succ x\rangle$, that correspond to the equality $x \approx y$; and the constrained rule

$$(\forall x)(\forall y)(f(x,y) \to f(y,x) \ll \langle id, f(x,y) \succ f(y,x)\rangle) \ ,$$

that is equivalent to

$$(\forall x)(\forall y)(f(x,y) \approx f(y,x)) \ .$$

The possibility to orient every equality justifies the following definition, that assigns to each set of equalities a constrained reduction system. Since it will be the starting point of the completion process, it is called the initial system:

Definition 2.38 *Let E be a set of equalities. Then*

$$\{(\forall \bar{x})(s \to t \ll \langle id, s \succ t\rangle) \mid (\forall \bar{x})(s \approx t) \in E \ or \ (\forall \bar{x})(t \approx s) \in E\}$$

is the initial *constrained reduction system assigned to E.*

A constrained reduction system $\mathcal{R}$ defines derivability relations $\Rightarrow_{\mathcal{R}}$ and $\Rrightarrow_{\mathcal{R}}$ on the set of constrained terms:

Definition 2.39 *Let $\mathcal{R}$ be a constrained reduction system and $\mathbf{t} = (\forall \bar{x})(t \ll c_t)$ a constrained term. Iff there is a rule $\mathbf{r} = (\forall \bar{y})(l \to r \ll c_r)$ in $\mathcal{R}$, such that*

1. *$\{x_1, \ldots, x_n\} \cap \mathrm{Var}(r) = \emptyset$ and $\{y_1, \ldots, y_m\} \cap \mathrm{Var}(t) = \emptyset$,*[14]
2. *p is a position in t where $t_{|p}$ is not a variable unless $t_{|p} = l = x_i$,*
3. *$t_{|p}$ and l are (syntactically) unifiable with an MGU ν,*
4. *the combination $c_{new} = \langle \mu, O_{new}\rangle = c_t \sqcap c_r \sqcap \langle \nu, \mathrm{true}\rangle$ is consistent,*

then $\mathbf{t} \Rightarrow_{\mathcal{R}} \mathbf{t}'$, where $\mathbf{t}' = (\forall \bar{x})(\forall \bar{y})((t[p/r])\mu \ll c_{new})$.[15]

Iff in addition (i) $t_{|p} = l\mu$, and (ii) c_{new} subsumes c_t, then $\mathbf{t} \Rrightarrow_{\mathcal{R}} \mathbf{t}'$. We call the triple $\langle \mathbf{r}, p, \mu\rangle$ a justification for $\mathbf{t} \Rightarrow_{\mathcal{R}} \mathbf{t}'$ (resp. $\mathbf{t} \Rrightarrow_{\mathcal{R}} \mathbf{t}'$).

The intuitive meaning of $(\forall \bar{x})(s \ll c_s) \Rightarrow_{\mathcal{R}} (\forall \bar{y})(t \ll c_t)$ is: there is a substitution σ such that $t\sigma$ can be derived from $s\sigma$ using a rule from $\mathcal{R}$, and σ satisfies the constraints c_s, c_t and that attached to the rule.

The main difference between the two derivability relations $\Rightarrow_{\mathcal{R}}$ and $\Rrightarrow_{\mathcal{R}}$ (which is a sub-relation of $\Rightarrow_{\mathcal{R}}$) is that the derivation $\mathbf{t} \Rrightarrow_{\mathcal{R}} \mathbf{t}'$ is "reversible", if the order on terms is not taken into concern. The derived term $\mathbf{t}'$ can—in combination with the rules in $\mathcal{R}$—take on the functions of $\mathbf{t}$. In contrary to that, a derivation $\mathbf{t} \Rightarrow_{\mathcal{R}} \mathbf{t}'$ is "irreversible" (provided $\mathbf{t} \not\Rrightarrow_{\mathcal{R}} \mathbf{t}'$).

[14] This is not a real restriction, since the bound variables can be renamed.

[15] If the constraint c_{new} expresses restrictions on *bound* variables that do not occur in $t[p/r]$, these restrictions can be omitted. For example, $(\forall x)(a \to b \ll \langle id, x \succ c\rangle)$ can be reduced to $a \to b \ll \epsilon$.

Example 2.40 *Some examples for derivations and their justification:*

$$\begin{array}{rcll}
(g(a,c) \ll \epsilon) & \Rrightarrow & (g(a,b) \ll \epsilon) & - \langle (c \to b \ll \epsilon),\, \langle 2 \rangle,\, id \rangle \\
(f(c) \ll \epsilon) & \Rrightarrow & (c \ll \epsilon) & - \langle ((\forall x)(f(x) \to x \ll \epsilon),\, \langle\rangle,\, id \rangle \\
(a \ll \epsilon) & \Rightarrow & (y \ll \langle \{x/a\}, a \succ y \rangle) & - \langle (x \to y \ll \langle id, x \succ y \rangle),\, \langle\rangle,\, \{x/a\} \rangle \\
(f(c) \ll \epsilon) & \Rightarrow & (c \ll \langle \{x/c\}, true \rangle) & - \langle (f(x) \to x \ll \epsilon),\, \langle\rangle,\, \{x/c\} \rangle
\end{array}$$

It is useful to define a subsumption relation on constrained terms. It is similar to the relation between a term (without constraint) and its instances:

Definition 2.41 *Let $\mathbf{t}_1 = (\forall \bar{x})(t_1 \ll c_1)$ and $\mathbf{t}_2 = (\forall \bar{y})(t_2 \ll c_2)$ be constrained terms. $\mathbf{t}_1$ subsumes $\mathbf{t}_2$ iff (i) t_2 is an instance of t_1, and (ii) the combination $c_1 \sqcap \langle \mu, true \rangle$ subsumes the constraint c_2 (Def. 2.35).*

Example 2.42 *The constrained term $a \ll \epsilon$ subsumes $a \ll \langle \{x \leftarrow a\}, true \rangle$. If $b \succ_{\mathrm{LPO}} a$, then the constrained rule $x \to a \ll \langle id, x \succ a \rangle$ subsumes the rule $b \to a \ll \langle \{x/b\}, true \rangle$.*

2.6.6.3 Completion of Constrained Reduction Systems

Goal of the Completion The following transformation rules define a method for completing constrained reduction systems. If these rules are applied repeatedly (in a fair way) to an initial system $\mathcal{R} = \mathcal{R}^0$, a system $\mathcal{R}^\infty$ is approximated. It represents the (classical) completions of all the different instances of E.

In general, the instances of $\mathcal{R}^\infty$ will not be irreducible and, therefore, not canonical. Nevertheless, the relation $\Rightarrow_{\mathcal{R}^\infty}$ will be confluent (in a sense clarified in Lemma 2.57), and thus have the feature crucial for computing normal forms of constrained terms and solving E-unification problems.

The following example shows that it would not make sense to expect the instances to be canonical:

Example 2.43 *None of the transformation rules introduced in the next section can be applied to the reduction system $\mathcal{R}^\infty = \{f(x) \to c \ll \epsilon,\ a \to b \ll \epsilon\}$. Its instance $\{f(a) \to c,\ a \to b\}$ is, nevertheless, not canonical, since it can be simplified to $\{f(b) \to c,\ a \to b\}$.*

The Transformation Rules The rules that have to be applied to complete a reduction system are presented in form of transformation rules.[16]

Deletion: A rule that has an inconsistent constraint attached to it can be removed, because it cannot be applied anyway:

$$\textbf{(Del)}\quad \frac{\mathcal{R} \cup \{(\forall \bar{x})(s \to t \ll c)\}}{\mathcal{R}} \qquad c \text{ inconsistent}$$

Example 2.44 *The rule $x \to f(x) \ll \langle id, x \succ f(x) \rangle$ can be deleted, because its constraint is inconsistent.*

Subsumption: A constrained rule that is subsumed by another rule (Def. 2.41) can be removed:

$$\textbf{(Sub)}\quad \frac{\mathcal{R} \cup \{\mathbf{r}, \mathbf{r}'\}}{\mathcal{R} \cup \{\mathbf{r}\}} \qquad \mathbf{r} \text{ subsumes } \mathbf{r}'$$

[16] The set of constrained rules below the line can be derived from the set above the line if the conditions on the right are met.

Equivalence Transformation: A constraint c attached to a reduction rule can be replaced by a set $\{c_1, \ldots, c_n\}$ of constraints that—disjunctively connected—are equivalent to c (i.e. $\mathrm{Sat}(c) = \bigcup_{1 \le i \le n} \mathrm{Sat}(c_i)$). Since only a single constraint can be attached to a rule, n copies of the original rule are generated:

$$\textbf{(Equ)} \quad \frac{\mathcal{R} \cup \{(\forall \bar{x})(l \to r \ll c)\}}{\mathcal{R} \cup \{(\forall \bar{x})(l\sigma \to r\sigma \ll \langle \sigma, O\rangle) \mid \langle \sigma, O\rangle \in C\}} \quad \begin{array}{l}\mathrm{Sat}(c) = \mathrm{Sat}(C),\\ C \text{ finite}\end{array}$$

Though this equivalence rule is not necessary for the completeness of our method, it is very useful; it allows to transform constraints into a normal form, and thus simplify their handling significantly.

Example 2.45 *The rule $f(x, y) \to f(a, b) \ll \langle id, f(x, y) \succ f(a, b)\rangle$ can be replaced by the two rules $f(x, y) \to f(a, b) \ll \langle id, x \succ a\rangle$ and $f(a, y) \to f(a, b) \ll \langle \{x/a\}, y \succ b\rangle$.*

Critical Pair Rule, Combination, Simplification: The transformation rules described so far allow to delete rules or to replace them by new ones without using the derivability relation $\Rightarrow_{\mathcal{R}}$. But, to complete a reduction system, $\Rightarrow_{\mathcal{R}}$ has to be taken into concern by applying one rule $\mathbf{r}_2 \in \mathcal{R}$ to another rule $\mathbf{r}_1 \in \mathcal{R}$. Suppose $\mathbf{r}_1 = (\forall \bar{x})(s \to t \ll c_1)$, $\mathbf{r}_2 = (\forall \bar{y})(l \to r \ll c_2)$, and the rule $\mathbf{r}_2$ can be applied to $\mathbf{r}_1$ to derive the rule $\mathbf{r}'_1 = (\forall \bar{x})(\forall \bar{y})(s_{new} \to t_{new} \ll c_{new})$, i.e., $\mathbf{r}_1 \Rightarrow \mathbf{r}'_1$ with a justification $\langle \mathbf{r}_2, p, \mu\rangle$. We cannot just add the new rule $\mathbf{r}'_1$ to $\mathcal{R}$: Firstly, instances of $\mathbf{r}'_1$ may be oriented differently; we therefore have to use the two symmetrical versions

$$\begin{aligned}\mathbf{r}_{new1} &= (\forall \bar{x})(\forall \bar{y})(s_{new} \to t_{new} \ll c_{new} \sqcap \langle id, s_{new} \succ t_{new}\rangle)\\ \mathbf{r}_{new2} &= (\forall \bar{x})(\forall \bar{y})(t_{new} \to s_{new} \ll c_{new} \sqcap \langle id, t_{new} \succ s_{new}\rangle)\ .\end{aligned}$$

Secondly, the form of the transformation rule depends on whether (i) $\mathbf{r}_1 \Rrightarrow \mathbf{r}'_1$ (besides $\mathbf{r}_1 \Rightarrow \mathbf{r}'_1$) or not,[17] and (ii) which side of $\mathbf{r}_1$ the rule $\mathbf{r}_2$ has been applied to, i.e., whether p is a position in s or in t.

If $\mathbf{r}_1 \Rrightarrow \mathbf{r}'_1$, then $\mathbf{r}_{new1}$ and $\mathbf{r}_{new2}$ allow—together with $\mathbf{r}_2$—all the derivations possible with $\mathbf{r}_1$. If, in addition, $\mathbf{r}_2$ has been applied to the right side of $\mathbf{r}_1$, one can conclude that the constraint attached to $\mathbf{r}_{new2}$ is inconsistent. In that case the transformation is called *simplification* (Sim), since $\mathbf{r}_1$ can be replaced by the single new rule $\mathbf{r}_{new1}$:

$$\textbf{(Sim)} \quad \frac{\mathcal{R}}{(\mathcal{R} \setminus \{\mathbf{r}_1\}) \cup \{\mathbf{r}_{new1}\}} \quad p \text{ in } t,\ \mathbf{r}_1 \Rrightarrow_{\mathcal{R}} \mathbf{r}'_1$$

Else, if $\mathbf{r}_2$ has been applied to the left side of $\mathbf{r}_1$, the rule $\mathbf{r}_{new2}$ cannot be left out, because the constraint attached to it may be consistent. Such a transformation is called *composition* (Com).

$$\textbf{(Com)} \quad \frac{\mathcal{R}}{(\mathcal{R} \setminus \{\mathbf{r}_1\}) \cup \{\mathbf{r}_{new1}, \mathbf{r}_{new2}\}} \quad p \text{ in } s,\ \mathbf{r}_1 \Rrightarrow_{\mathcal{R}} \mathbf{r}'_1$$

If $\mathbf{r}_1 \not\Rrightarrow \mathbf{r}'_1$, the new rules cannot replace the old rule $\mathbf{r}_1$; it cannot be removed. Nevertheless, the transformation has to be carried out provided $\mathbf{r}_2$ has been applied to the left side of $\mathbf{r}_1$. Then $\mathbf{r}_1$ and $\mathbf{r}_2$ are a *critical pair*, and the new rules are needed to make the reduction system confluent:

$$\textbf{(CP)} \quad \frac{\mathcal{R}}{\mathcal{R} \cup \{\mathbf{r}_{new1}, \mathbf{r}_{new2}\}} \quad p \text{ in } s,\ \mathbf{r}_1 \not\Rrightarrow_{\mathcal{R}} \mathbf{r}'_1$$

In difference to the critical pair rule defined in (Gallier *et al.*, 1992) the unifier μ is only applied locally to the new rules (not to the whole system $\mathcal{R}$).

[17] That is, $\mathbf{r}_1 \Rrightarrow \mathbf{r}'_1$ with the same justification as $\mathbf{r}_1 \Rightarrow \mathbf{r}'_1$; whether $\mathbf{r}_1 \Rrightarrow \mathbf{r}'_1$ with a different justification is not relevant.

Example 2.46 *Suppose $f \succ_{\text{LPO}} c \succ_{\text{LPO}} b \succ_{\text{LPO}} a$, and $\mathcal{R}$ contains the constrained reduction rules*

$$\begin{array}{rclrcl} \mathbf{r}_1 & = & f(c) \to b \ll \epsilon & \mathbf{r}_3 & = & (\forall x)(f(x) \to y \ll \langle id, f(x) \succ y \rangle) \\ \mathbf{r}_2 & = & b \to a \ll \epsilon & \mathbf{r}_4 & = & f(x) \to y \ll \langle id, f(x) \succ y \rangle \end{array}$$

The simplification rule (Sim) can be applied to $\mathbf{r}_1$ and $\mathbf{r}_2$ to replace $\mathbf{r}_1$ by the single new rule $f(c) \to a \ll \epsilon$.

The composition rule (Com) can be applied to $\mathbf{r}_1$ and $\mathbf{r}_3$ to replace $\mathbf{r}_1$ by

$$y \to b \ll \langle id, (f(c) \succ y \wedge y \succ b) \rangle \ \text{ and } \ b \to y \ll \langle id, (f(c) \succ y \wedge b \succ y) \rangle \ .$$

The critical pair rule (CP) can be applied to $\mathbf{r}_1$ and $\mathbf{r}_4$ (note, that in $\mathbf{r}_4$ the variable x is not quantified); the new rules

$$y \to b \ll \langle \{x/c\}, (f(c) \succ y \wedge y \succ b) \rangle \ \text{ and } \ b \to y \ll \langle \{x/c\}, (f(c) \succ y \wedge b \succ y) \rangle$$

have to be added.

Fair Completion Procedures In general, an infinite number of transformation steps can be necessary to complete a reduction system. But even if the computation does not terminate, a completion $\mathcal{R}^\infty$ is approximated, consisting of the *persistent* reduction rules, that occur in all but a finite number of the resulting system. To generate a confluent reduction systems, certain fairness conditions have to be met:

Definition 2.47 *$\mathcal{R} \vdash \mathcal{R}'$ means that the constrained reduction system $\mathcal{R}'$ can be derived from $\mathcal{R}$ by applying one of the transformation rules from Section 2.6.6.3.*

A transformation procedure *specifies, when supplied with an initial reduction system $\mathcal{R}^0$, in which way (in particular: in which order) the transformation rules are to be applied to generate a sequence $\mathcal{R}^0 \vdash \mathcal{R}^1 \vdash \mathcal{R}^2 \vdash \cdots$ of reduction systems. Then, the reduction system*

$$\mathcal{R}^\infty = \begin{cases} \mathcal{R}^m & \text{if the sequence is of length } m \\ \bigcup_{k \geq 0} \bigcap_{m \geq k} \mathcal{R}^m & \text{if the sequence is infinite} \end{cases}$$

is called the completion *of $\mathcal{R} = \mathcal{R}^0$, and the completion of the set E of equalities if $\mathcal{R}$ is the initial system for E.*

A transformation procedure is fair *provided:*

1. *There is no infinite sequence $(\mathbf{r}_i)_{i \geq 0} \subset \bigcup_{m \geq k} \mathcal{R}^m$ such that for all $i \geq 0$ the rule $\mathbf{r}_{i+1}$ has been derived from $\mathbf{r}_i$ by an equivalence transformation.*
2. *There is no infinite sequence $(\mathbf{r}_i)_{i \geq 0} \subset \bigcup_{m \geq k} \mathcal{R}^m$ such for all $i \geq 0$ the rule $\mathbf{r}_{i+1}$ subsumes $\mathbf{r}_i$, and $\mathbf{r}_i$ has therefore been removed.*
3. *For every persistent critical pair $\mathbf{r}_1, \mathbf{r}_2 \in \mathcal{R}^\infty$ there is an $i \geq 0$ such that $\mathcal{R}^{i+1}$ has been derived by applying the critical pair transformation rule to $\mathbf{r}_1, \mathbf{r}_2 \in \mathcal{R}^i$.*

The first two fairness conditions are of a more technical nature: Condition 1 avoids infinite sequences of equivalence transformations. Condition 2 assures that, if there is an infinite sequence of rules subsuming each other, at least one of them is in the completion $\mathcal{R}^\infty$.

Condition 3 is the most important: it assures the application of the critical pair transformation rule to all persistent critical pairs. It is essential for achieving confluence of the completion.

Provided, the above fairness conditions are met, arbitrary heuristics can be used to choose the next transformation rule to apply.

2.6.6.4 Computing Normal Forms

Normalization Rules Using constrained reduction systems and terms, a term has more than one normal form—in general an infinite number of them.

Example 2.48 *With $\mathcal{R}^\infty = \{b \to a \ll \epsilon,\ d \to c \ll \epsilon\}$ the constrained term $x \ll \epsilon$ has three normal forms: $a \ll \langle\{x/b\}, true\rangle$, $c \ll \langle\{x/d\}, true\rangle$, and $x \ll \epsilon$ itself.*

The above example shows that there can be redundancies in a set of normal forms: the validity of $x \ll \epsilon$ is not restricted to substitutions σ such that $\sigma(x) \neq a$ and $\sigma(x) \neq b$.

The computation of normal forms is—similar to the completion procedure—presented in form of transformation rules operating on sets of constrained terms:

Definition 2.49 *To compute the normal forms of a set $\mathcal{T}$ of constrained terms, the rules* **deletion** *(Del),* **equivalence** *(Equ),* **subsumption** *(Sub),* **simplification** *(Sim), and* **deduction** *(Ded) can be applied to $\mathcal{T}$; the rules depend on a constrained reduction system $\mathcal{R}$:*

$$\textbf{(Del)}\quad \frac{\mathcal{T} \cup \{(\forall\bar{x})(t \ll c)\}}{\mathcal{T}} \qquad c \text{ inconsistent}$$

$$\textbf{(Equ)}\quad \frac{\mathcal{T} \cup \{(\forall\bar{x})(t \ll c)\}}{\mathcal{T} \cup \{(\forall\bar{x})(t\sigma \ll \langle\sigma, O\rangle) \mid \langle\sigma, O\rangle \in C\}} \qquad \mathrm{Sat}(c) = \mathrm{Sat}(C),\ C \text{ finite}$$

$$\textbf{(Sub)}\quad \frac{\mathcal{T} \cup \{\mathbf{t}, \mathbf{t}'\}}{\mathcal{T} \cup \{\mathbf{t}\}} \qquad \mathbf{t} \text{ subsumes } \mathbf{t}'$$

$$\textbf{(Sim)}\quad \frac{\mathcal{T} \cup \{\mathbf{t}\}}{\mathcal{T} \cup \{\mathbf{t}'\}} \qquad \mathbf{t} \Rrightarrow_{\mathcal{R}} \mathbf{t}'$$

$$\textbf{(Ded)}\quad \frac{\mathcal{T} \cup \{\mathbf{t}\}}{\mathcal{T} \cup \{\mathbf{t}, \mathbf{t}'\}} \qquad \mathbf{t} \Rightarrow_{\mathcal{R}} \mathbf{t}',\ \mathbf{t} \not\Rrightarrow_{\mathcal{R}} \mathbf{t}'$$

Fair Normalization Procedures As for completion, an infinite number of normalization steps can be necessary; similar fairness conditions have to be met. A set $\mathcal{T}^\infty$ of normal forms is approximated, consisting of the *persistent* terms, that occur in all but a finite number of the sets.

Definition 2.50 *$\mathcal{T} \vdash \mathcal{T}'$ means that the set $\mathcal{T}'$ of constrained terms can be derived from $\mathcal{T}$ by applying one of the normalization rules from Definition 2.49.*

A normalization procedure *specifies, when supplied with an initial set $\mathcal{T}^0$ of constrained terms and a reduction system $\mathcal{R}$, in which way the rules are to be applied to generate a sequence $\mathcal{T}^0 \vdash \mathcal{T}^1 \vdash \mathcal{T}^2 \vdash \cdots$ of sets of constrained terms. Then, the set*

$$\mathcal{T}^\infty = \begin{cases} \mathcal{T}^m & \text{if the sequence is of length } m \\ \bigcup_{k\geq 0}\bigcap_{m\geq k}\mathcal{T}^m & \text{if the sequence is infinite} \end{cases}$$

is called the set of normal forms *of $\mathcal{T} = \mathcal{T}^0$ (w.r.t. $\mathcal{R}$).*

A normalization procedure is fair *provided:*

1. *There is no infinite sequence $(\mathbf{t}_i)_{i\geq 0} \subset \bigcup_{m\geq k}\mathcal{T}^m$ such that for all $i \geq 0$ the term $\mathbf{t}_{i+1}$ has been derived from $\mathbf{t}_i$ by an application of equivalence (Equ).*

2. *There is no infinite sequence* $(\mathbf{t}_i)_{i\geq 0} \subset \bigcup_{m\geq k} \mathcal{T}^m$ *such that for all* $i \geq 0$ *the term* $\mathbf{t}_{i+1}$ *subsumes* $\mathbf{t}_i$*, and* $\mathbf{t}_i$ *has therefore been removed.*

3. *For every persistent term* $\mathbf{t} \in \mathcal{T}^\infty$ *that a rule* $\mathbf{r} \in \mathcal{R}$ *can be applied to, there is an* $i \geq 0$ *such that* $\mathcal{T}^{i+1}$ *has been derived by applying* $\mathbf{r}$ *to* $\mathbf{t} \in \mathcal{T}^i$*.*

The first two fairness conditions are similar to that of fair completion procedures (Def. 2.47). Condition 3 assures that whenever possible deduction and simplification are applied to persistent terms.

Combining Completion and Normalization Although a completion $\mathcal{R}^\infty$ may be infinite, one has to abandon the computation of further reduction rules at a certain point, if completion and normalization of terms are separated. It is very difficult to decide when this point is reached. Therefore, it is better to combine the completion and the normalization process:

Definition 2.51 *A* completion and normalization sequence $(\langle\mathcal{R}^i, \mathcal{T}^i\rangle)_{i\geq 0}$ *consists of constrained reduction systems* $\mathcal{R}^i$ *and sets* $\mathcal{T}^i$ *of constrained terms, where (for* $i \geq 0$*) either (i)* $\mathcal{R}^{i+1}$ *has been derived from* $\mathcal{R}^i$ *by applying a transformation rule (Sec. 2.6.6.3) and* $\mathcal{T}^i = \mathcal{T}^{i+1}$*; or (ii)* $\mathcal{T}^{i+1}$ *has been derived from* $\mathcal{T}^i$ *by applying a normalization rule (Def. 2.49) and* $\mathcal{R}^i = \mathcal{R}^{i+1}$*.*

Of course, when completion and normalization are combined, the fairness conditions (Def. 2.47 and 2.50) still have to be met.

2.6.6.5 Solving an E-Unification Problem

Now we can solve an arbitrary mixed E-unification problem $\langle E, s, t\rangle$ by completing the initial reduction system $\mathcal{R}^0$ for E and computing the sets of normal forms of the constrained terms $s \ll \epsilon$ and $t \ll \epsilon$. Using these normal forms, sets $\mathcal{C}^i$ of constraints can be computed that are satisfied by solutions to the unification problem. These approximate a set $\mathcal{C}$ such that $\mathrm{Sat}(\mathcal{C})$ is a complete set of unifiers:

Definition 2.52 *Let* $\langle E, s, t\rangle$ *be a mixed E-unification problem,* $\mathcal{R}^0$ *the initial system for E,* $\mathcal{S}^0 = \{s \ll \epsilon\}$*,* $\mathcal{T}^0 = \{t \ll \epsilon\}$*, and* $(\langle\mathcal{R}^i, \mathcal{S}^i\rangle)_{i\geq 0}$ *and* $(\langle\mathcal{R}^i, \mathcal{T}^i\rangle)_{i\geq 0}$ *fair completion and normalization procedures. Then, for* $(i = 0, 1, 2, \ldots, \infty)$ *the sets* $\mathcal{C}^i(\langle E, s, t\rangle)$ *consist of the constraints*

$$\{c_1 \sqcap c_2 \sqcap \langle \mu, \mathit{true}\rangle \mid (\forall \bar{x})(r_1 \ll c_1) \in \mathcal{S}^i,\ (\forall \bar{y})(r_2 \ll c_2) \in \mathcal{T}^i, \\ r_1 \text{ and } r_2 \text{ are (syntactically) unifiable with an MGU } \mu\ \}$$

$\mathcal{C}(\langle E, s, t\rangle)$ *denotes their union* $\bigcup_{i\geq 0} \mathcal{C}^i(\langle E, s, t\rangle)$*.*

2.6.6.6 Soundness, Completeness, Confluence

In this section we state soundness and completeness results for our method. Due to space restrictions the proofs are omitted; they can be found in (Beckert, 1993b).

Theorem 2.53 (Soundness) *Let* $\langle E, s, t\rangle$ *be a mixed E-unification problem. A substitution* σ *satisfying one of the constraints in* $\mathcal{C}(\langle E, s, t\rangle)$ *(Def. 2.52) is a solution to* $\langle E, s, t\rangle$*.*

Since our aim is to find *most general* unifiers (MGUs), a subsumption relation on substitutions has to be defined. One could use the specialization relation $\leq$. But, for solving mixed E-unification problems, the subsumption relation $\leq_E$ is better suited:[18]

[18] A similar subsumption relation—for purely rigid problems—has been defined in (Gallier *et al.*, 1992).

Definition 2.54 *Let E be a set of equalities. The subsumption relation $\leq_E$ is defined on the set of substitutions by: $\sigma \leq_E \tau$ iff there is a substitution σ' such that $E\tau \models (\sigma' \circ \sigma)(x)\mathrm{Gl}(\tau)(x)$ for all variables x, where the free variables in $E\tau$ are held rigid.*

Theorem 2.55 (Completeness) *Let $\langle E, s, t\rangle$ be a mixed E-unification problem. The set*

$$\mathrm{Sat}(\mathcal{C}(\langle E, s, t\rangle))$$

of unifiers is ground-complete w.r.t. the subsumption relation $\leq_E$ (Def. 2.54), i.e., for every ground unifier σ of $\langle E, s, t\rangle$ there is a substitution $\tau \in \mathrm{Sat}(\mathcal{C}(\langle E, s, t\rangle))$ such that $\tau \leq_E \sigma$.

A ground-complete set of unifiers w.r.t. the relation $\leq$ can be computed by inverting the constrained rules in a completion $\mathcal{R}^\infty$ for E (i.e., by changing their orientation, not the validity of their constraints), and applying the inversion to the unifiers in $\mathrm{Sat}(\mathcal{C}(\langle E, s, t\rangle))$. Computing these additional solutions can be necessary—in theory—to find solutions to a simultaneous E-unification problem by combining solutions to its components. Fortunately, in practice this turns out to be very rarely the case, in particular in the semantic tableau framework.

$\overset{*}{\Rightarrow}_{\mathcal{R}^\infty}$ is in general not well founded. Therefore, our method is only a semi-deciding procedure for unifiability—even if the completion $\mathcal{R}^\infty$ is finite (it is an open problem, whether $\overset{*}{\Rightarrow}_{\mathcal{R}^\infty}$ is well founded for purely rigid E-unification problems). The following example shows that, in addition, $\overset{*}{\Rightarrow}_{\mathcal{R}^\infty}$ cannot be expected to be confluent:

Example 2.56 *Supposed there are rules $f(a) \to a \ll \epsilon$ and $f(b) \to b \ll \epsilon$ in $\mathcal{R}^\infty$. Then from the constrained term $\mathbf{s} = f(x) \ll \epsilon$ terms $\mathbf{t}_1 = a \ll \langle\{x/a\}, \mathit{true}\rangle$ and $\mathbf{t}_2 = b \ll \langle\{x/b\}, \mathit{true}\rangle$ can be derived (i.e. $\mathbf{s} \Rightarrow_{\mathcal{R}^\infty} \mathbf{t}_1$ and $\mathbf{s} \Rightarrow_{\mathcal{R}^\infty} \mathbf{t}_2$).*

If $\Rightarrow_{\mathcal{R}^\infty}$ were confluent, there would have to be a term derivable from both $\mathbf{t}_1$ and $\mathbf{t}_2$. That would not make any sense but contradicts soundness.

However, the derivability relation $\overset{*}{\Rightarrow}_{\mathcal{R}^\infty}$ can be proven to be "weak" confluent (the proof of Theorem 2.55 is based upon that):

Lemma 2.57 *If $\mathcal{R}^\infty$ is a fair completion, $\mathbf{s}$, $\mathbf{t}_1$ and $\mathbf{t}_2$ are constrained terms such that*

1. *$\mathbf{s} \overset{*}{\Rightarrow}_{\mathcal{R}^\infty} \mathbf{t}_1$ and $\mathbf{s} \overset{*}{\Rightarrow}_{\mathcal{R}^\infty} \mathbf{t}_2$,*
2. *the combination $c_1 \sqcap c_2$ is consistent,*

then there are constrained terms $\mathbf{u}_1$ and $\mathbf{u}_2$, such that

1. *$\mathbf{t}_1 \overset{*}{\Rightarrow}_{\mathcal{R}^\infty} \mathbf{u}_1$ and $\mathbf{t}_2 \overset{*}{\Rightarrow}_{\mathcal{R}^\infty} \mathbf{u}_2$,*
2. *$\mathbf{u}_1$ and $\mathbf{u}_2$ have a common instance.*

2.7 Many-Valued Tableaux Using Sets-As-Signs

A closer inspection of the proof trees in Figure 2.2 reveals that all unsigned formulae in the tree on the left occur also in the tree on the right and at the same position: the tree on the left is isomorphic to a subtree of the tree on the right. Inspection of other examples shows that there is always a very high degree of redundancy in the trees corresponding to the various non-designated truth values.

Consider, for example, the signed formula $2 \sim \phi$. Application of the corresponding tableau rule after Surma and Carnielli yields two new branches containing the formulae $0\,\phi$ and $1\,\phi$ respectively. Encounter of such a formula during a proof, however, does not give rise to any logical reason to split the proof tree at once into the two cases determined by the extensions of the rule. If we were able to express the more complex assertion *ϕ has either truth value* 0 *or truth value* 1 with a single signed formula we could avoid the splitting. Hence, our idea is to increase the expressivity of the sign language in order to be able to state more complex conditions like that. Perhaps the most natural thing to do is to admit *subsets of the set of truth values* as signs.

Definition 2.58 (Base Set of Signs) *Let* $\bar{\mathsf{S}} = \{\{\mathsf{k}_1, \ldots, \mathsf{k}_m\} \mid \{k_1, \ldots, k_m\} \subseteq N\} = 2^{\mathsf{N}}$ *be the* **base set of signs***.*

Here we assume that the set of signs $\mathbf{S}_{\mathcal{L}}$ in a logic $\mathcal{L}$ always obeys

$$\{\{1\}, \ldots, \{\mathsf{k}\}\} \subseteq \mathbf{S}_{\mathcal{L}} \subseteq \bar{\mathsf{S}} \tag{2.2}$$

We need the left part of equation (2.2), because otherwise unsound rules can be stated. In (Hähnle, 1992c) a more general condition that is sufficient for soundness is given.

Example 2.59 *Let us fix the set of signs as*

$$\mathsf{S} = \{\{0\}, \{1\}, \{2\}, \{0,1\}, \{1,2\}\}$$

We can express the assertion that ϕ has either truth value 0 or truth value 1 *with the signed formula $\{0,1\}\,\phi$. An equivalent formulation would be to say that ϕ cannot take on truth value 2, hence we need to build only one tableau proof tree for each proof.*

To compute the tableau rules corresponding to signed formulae of the form $\mathsf{S}\,\phi$ we must, similar as before, find a cover for all entries in the truth table of F that are a member of S and then minimize the resulting expression. The difference to the former approach is that extensions in which formulae with generalized signs do occur cover more than one entry in general.

For example, to compute the rule with premise $\{0,1\}\,(\phi \wedge \psi)$ we have to cover all entries in the truth table of $\wedge$. The minimal rule that does the job would be

$$\frac{\{0,1\}\,(\phi \wedge \psi)}{\{0,1\}\,\phi \;\Big|\; \{0,1\}\,\psi}$$

The rule for $\{1\}\,(\phi \wedge \psi)$ would be

$$\frac{\{1\}\,(\phi \wedge \psi)}{\begin{array}{c|c} \{1,2\}\,\phi & \{1\}\,\phi \\ \{1\}\,\psi & \{1,2\}\,\psi \end{array}}$$

which is much simpler than Carnielli's rule for the same premise on page 20.

Example 2.60 *We show that $\vdash_{\mathsf{S}} \neg p \supset (\sim p \wedge \neg p)$ holds in the tableau system corresponding to the signs from Example 2.59 (it is an instructive exercise to compute the missing tableau rules). The same fact was proven in Example 2.15. Note that the single tree required now has exactly the size of the smaller of the two trees before.*

$$\begin{tikzcd}
 & (1)\ [-]\ \ \{0,1\}\,(\neg p \supset (\sim p \wedge \neg p)) \arrow[d, no head] & \\
 & (2)\ [1]\ \ \{2\}\,\neg p \arrow[d, no head] & \\
 & (3)\ [1]\ \ \{0,1\}\,(\sim p \wedge \neg p) \arrow[d, no head] & \\
 & (4)\ [2]\ \ \{0\}\,p \arrow[dl, no head] \arrow[dr, no head] & \\
(5)\ [3]\ \ \{0,1\} \sim p \arrow[d, no head] & & (7)\ [3]\ \ \{0,1\}\,\neg p \arrow[d, no head] \\
(6)\ [5]\ \ \{2\}\,p & & (8)\ [7]\ \ \{1,2\}\,p \\
\textit{closed with } (4,6) & & \textit{closed with } (4,8)
\end{tikzcd}$$

Note, however, that minimal tableau rules using sets-as-signs need not to be unique (see (Hähnle, 1992c) for examples).

Quantifier rules for many-valued logic are surprisingly simple if only certain signs are used. We define the following abbreviations for signs:

$$\begin{array}{rcl}
\boxed{<\mathsf{i}} & = & \{\mathsf{0}\,,\ldots,\mathsf{i}-1\} \text{ for } i \in N \\
\boxed{>\mathsf{i}} & = & \{\mathsf{i}+1,\ldots,\mathsf{n}-1\} \text{ for } i \in N
\end{array}$$

For these signs we get the usual γ- and δ-rules, with the component rules as stated in Table 2.7.

If we ask for singleton sets-as-signs we get the rules from Table 2.6. Note, however, that *both* rules for singleton signs must be applied an indefinite number of times to its premise in order to guarantee completeness. In this sense, they have a γ-flavour, although they are not of γ shape.

$$\frac{\{\mathsf{i}\}\ (\forall x)\phi(x)}{\begin{array}{c}\boxed{>\mathsf{i}-1}\,\phi(t)\\ \{\mathsf{i}\}\ \phi(c)\end{array}} \qquad\qquad \frac{\{\mathsf{i}\}\ (\exists x)\phi(x)}{\begin{array}{c}\boxed{<\mathsf{i}+1}\,\phi(t)\\ \{\mathsf{i}\}\ \phi(c)\end{array}}$$

Where c is a new parameter and t is any term.

Table 2.6: Tableau rules for quantifiers with singleton signs.

Summary

Theorem 2.61 (Completeness) *Let ϕ be a first-order formula. If ϕ is a tautology then there is a closed tableau with root $\mathsf{N}-\mathsf{D}\phi$, provided for the set of signs S used in the tableau rules holds that $N-D \in S \subseteq \bigcup_{i\in N}\{\boxed{>\mathsf{i}},\boxed{<\mathsf{i}},\{\mathsf{i}\}\}$ and (2.2). If ϕ is propositional we need only condition (2.2).*

γ	$\gamma(t)$
$\boxed{>\mathsf{i}}\,(\forall x)\phi(x)$	$\boxed{>\mathsf{i}}\,\phi(t)$
$\boxed{<\mathsf{i}}\,(\exists x)\phi(x)$	$\boxed{<\mathsf{i}}\,\phi(t)$

δ	$\delta(t)$
$\boxed{<\mathsf{i}}\,(\forall x)\phi(x)$	$\boxed{<\mathsf{i}}\,\phi(c)$
$\boxed{>\mathsf{i}}\,(\exists x)\phi(x)$	$\boxed{>\mathsf{i}}\,\phi(c)$

Table 2.7: γ and δ component rules.

Theorem 2.62 (Soundness) *Let ϕ be a first-order formula. If there is a closed tableau with root $\boxed{\mathsf{N}-\mathsf{D}}\,\phi$ then ϕ is a tautology, provided (2.2) holds for the set of signs S used in the tableau rules.*

3 Syntax of Knowledge Bases

3.1 Parts of a Knowledge Base

$_3T^AP$'s input files (called knowledge bases) consist of four major parts: the sort declarations, signature definitions, the axioms, and the theorems. These are described in the following section. Comments (Section 3.5) may be inserted at any place into an input file.

A formal definition of $_3T^AP$'s input language is given in Table 3.1. For ease of understanding it is presented by a mixture of regular expressions and a context-free grammar (similar to extended Backus-Naur-form). Lowercase words denote lexems (terminals) and upper case words are used for non-terminals. Characters in quotes, e.g. '<', stand for themselves. Anything within square brackets [...] is optional. The operator * is the Kleene-hull, i.e. a^* could be expanded to the empty word or any finite sequence of as.

3.2 Sorts and Sort Declarations

The sort declaration may be omitted in the propositional case. For one-sorted first-order problems at least one sort has to be defined, e.g. `top`. The `top` sort is recommended for that case since the output utilities, which may be used to visualize a $_3T^AP$ proof, supply an option to omit that sort.

Remark 3.1 *Please note, that the* `top` *sort is* not predefined. *It has to be declared like any other sort.*

The <-notation is used to define a sort hierarchy. The symbol < may be read as "is subsort of". The sortname on the left side of < must not already be defined, the sortname on the right side must be defined. Otherwise the compiler will print an error message.

From a theoretical point of view it would be possible to use a finite meet-semilattice of sorts since a unique most general unifier exists for such hierarchies. In $_3T^AP$, however, we are restricted to tree-shaped sort hierarchies.

Example 3.2 *The following declaration defines the sort hierarchy shown in Figure 3.1.*

```
sort fields.
sort real < fields.
sort complex < fields.
sort rational < real.
```

KNOWLEDGE_BASE	::=	(DECLARATION)*
		(AXIOMS_AND_THEOREMS)*
DECLARATION	::=	SORT_DECLARATION
	|	PREDICATE_DECLARATION
	|	CONSTANT_DECLARATION
	|	FUNCTION_DECLARATION
	|	VARIABLE_DECLARATION
	|	COMMENT
SORT_DECLARATION	::=	’sort’ SORTNAME (’,’ SORTNAME)*
		[’<’ SORTNAME] ’.’
SORTNAME	::=	’top’ | NAME
PREDICATE_DECLARATION	::=	’predicate’ NAME (’,’ NAME)* [’:’ DOMAIN]’.’
DOMAIN	::=	SORTNAME (’x’ SORTNAME)* ’.’
CONSTANT_DECLARATION	::=	’constant’ NAME (’,’ NAME)* : SORTNAME ’.’
FUNCTION_DECLARATION	::=	’function’ NAME (’,’ NAME)* : DOMAIN
		’->’ SORTNAME ’.’
VARIABLE_DECLARATION	::=	’variable’ NAME (’,’ NAME)* : SORTNAME ’.’
AXIOMS_AND_THEOREMS	::=	’axiom’ NAME ’;’ FORMULA ’.’
	|	’theorem’ NAME ’;’ FORMULA ’.’
	|	COMMENT
FORMULA	::=	ATOMIC_FMA
	|	FORMULA BINOP FORMULA
	|	UNOP FORMULA
	|	’(’ FORMULA ’)’
	|	QUANTIFIER VARIABLES ’(’ FORMULA ’)’
ATOMIC_FMA	::=	NAME [’(’ TERM (’,’ TERM)* ’)’]
	|	TERM ’=’ TERM | TERM ’==’ TERM
TERM	::=	NAME [’(’ TERM (’,’ TERM)* ’)’]
VARIABLES	::=	NAME (’,’ NAME)* [’:’ SORTNAME]
QUANTIFIER	::=	’forall’ | ’exists’
BINOP	::=	’&’ | ’v’ | ’<=’ | ’<=>’ | ’=>’
	|	’<-’ | ’<->’ | ’->’ | ’<~’ | ’<~>’ | ’~>’ | ’#’
UNOP	::=	’-’ | ’~’ | ’jt’ | ’jf’ | ’ju’ | ’jt’ | ’aff’ | ’nabla’
COMMENT	::=	’%’ ALPHANUM*
NAME	::=	LOWERCASE ALPHANUM*
LOWERCASE	::=	’a’ | . . . | ’z’
UPPERCASE	::=	’A’ | . . . | ’Z’
OTHER_CHAR	::=	’0’ | . . . | ’9’ | ’_’
ALPHANUM	::=	LOWERCASE | UPPERCASE | OTHER_CHAR

Table 3.1: Definition of $_3T^AP$’s input language.

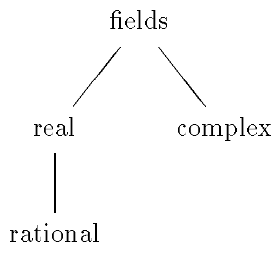

Figure 3.1: The sort hierarchy from Example 3.2.

3.3 Signature Definition

Every predicate, function, constant or variable symbol used in some axiom or theorem has to be defined first to establish its domain and—in the function case—its range.

Example 3.3 `sort top.`

```
predicate p.
predicate q1, q2 : top.
predicate r      : top x top.
function  f,g    : top x top -> top.
constant  c      : top.
variable  x      : top.
```

The above definition declares `p` *as propositional variable (predicate of arity 0);* `q1` *and* `q2` *are predicates with one argument of sort* `top`*; and* `r` *is a predicate of arity 2 with both arguments of sort* `top`*.* `f` *and* `g` *are declared as* `top`*-valued functions with two arguments of the same sort.* `c` *is a constant of sort* `top` *and* `x` *is a* `top`*-valued variable.*

Remark 3.4 *It is not possible to declare functions of arity 0. That is what constants are for. If you try this, the compiler will respond with an error message such as*

```
ERROR: line 10 near '->' : parse error
```

Remark 3.5 *$_3T^AP$ is not able to distinguish functions by their arity, i.e., you must not declare functions of different arity with the same name. If you do, the compiler will print an error message:*

```
ERROR: Redeclaration of 'f'! 'f' is already declared as a function!
```

The same is true for predicate declarations.

In addition, it is not allowed to give the same name to different types of symbols (e.g. a variable and a sort).

A symbol has to be defined before it is used. There is no other restriction on the order of the declarations.

The declaration of variables is an exception: Variables must either be defined by a variable declaration (as described above) or in the variable list following the quantifier. For example,

`v`	disjunction
`&`	conjunction
`=>`	material implication
`<=`	reverse material implication
`<=>`	equivalence
`-`	negation

Table 3.2: Two-Valued connectives.

```
...forall x:top...
```

has the same effect as

```
variable x:top.
...forall x...
```

However, something like

```
variable x:top.
...forall x:bottom...
```

and even

```
variable x:top.
...forall x:top...
```

is not allowed. If you try this, the compiler will respond with

```
ERROR: Redeclaration of 'x'! 'x' is already declared as a variable !
```

3.4 Axioms and Theorems

3.4.1 Names

Similar to signature declarations, axiom and theorem names have to be unique. The axiom and theorem names may be identical to names of sorts, predicates, functions, constants or variables; but this should be avoided because it is no good style.

3.4.2 Connectives

In the (standard) two-valued version of $_3T^AP$ the connectives in Table 3.2 are available. For the three-valued version the connectives from Table 3.3 may be used. Those connectives listed in the grammar but not listed in some of the tables have no predefined semantics. They may be freely defined to have any desired semantics. How to define operators is described in Section 9.3.

The connectives' priorities are defined in Table 3.4. Symbols in one box have the same priority. Please note that a higher priority means a lower precedence. For example, implication has a higher priority than disjunction (the precedence of disjunction is greater), i.e., the meaning of `a v b => c` is $(a \vee b) \rightarrow c$.

`v`	disjunction
`&`	conjunction
`=>`	weak implication
`<=`	reverse weak implication
`<=>`	weak equivalence
`#`	Lukasiewicz implication
`-`	strong negation
`~`	weak negation
`nabla`	(∇) nabla operator
`jt`	affirmation
`aff`	partial affirmation
`ju`	partial affirmation
`jf`	falsification

Table 3.3: Three-valued connectives.

max. priority
`<=>`

`#`
`~> <~> <~`
`-> <-> <-`
`=> <=`

`v`
`&`

`- ~`
`aff jf ju jt nabla`

`= ==`
min. priority

Table 3.4: Priority of the connectives.

3.4.3 Equality

As indicated by the grammar in Table 3.1, the equality sign = is a predefined infix predicate symbol. It must not be declared.

Make sure that the two sides of an equality have compatible sorts. In the following example `theorem t1` is illegal (in the scope of the sort declaration from Example 3.2), while `theorem t2` is correct:

```
variable x : real.
variable y : complex.
variable r : rational.
theorem t1; x = y.
theorem t2; x = r.
```

Remark 3.6 *If incompatible sorts are used in an equality, the compiler will print an error message. For the above example the error message is the following:*

```
ERROR: The terms 'x:[real,fields|_5001]' and 'y:[complex,fields|_5002]'
       have incompatible types!
Operator: =
```

The demodulator sign `==` may appear anywhere where an equality sign is allowed. The same restrictions concerning sort compatibility apply. More on demodulators can be found in Section 5.13.

3.5 Comments

Comments may be inserted into input files between any lexems. As indicated by the grammar, anything following the `%`-sign in a line is a comment.

To preserve readability, only two comment-rules are present in the grammar of Table 3.1. The non-terminal "COMMENT" may be inserted between any two symbols in the grammar. To preserve the readability of the input file, you should not make use of the additional places allowed for comments. It is a better style to comment formulae immediately before their definitions.

4 An Overview of the System Architecture

In this chapter we want to give the reader an idea of the general architecture of $_3T^AP$ and of how the modules interact.

We have already mentioned that $_3T^AP$ is mainly written in Quintus (resp. SICStus) Prolog with small pieces of C for efficient management of global variables.

Theorem proving in $_3T^AP$ is essentially a two stage process. One or more problems are collected into knowledge bases (see Chapter 3 for the syntax). In a first step these are compiled. The result is a representation of the problem in terms of $_3T^AP$'s internal data structures together with static link information. This is written to a separate file and thus needs to be computed only once. The second step is the actual proof.

The design of $_3T^AP$ tries to provide as much modularity as possible. The main tasks of the prover have been identified and were distributed in various modules. To understand their structure it will be helpful to look at the slightly simplified algorithm corresponding to the main predicate for producing a closed tableau (see Section 5.2 for more details):[1]

Algorithm of `close_multiple`
input: the current branch B and the conclusion C produced from the last rule application.
output: **success** iff all extensions E_i in C can be closed with B, **fail** else.

```
if   $B \cup E_1$ can be closed immediately
then success
else fetch one or more new formulae $\Phi$ from the knowledge base;
     if     $B \cup E_1 \cup \Phi$ can be closed immediately
     then   success
     elseif $B \cup E_1 \cup \Phi$ is exhausted
     then   if   $B \cup E_1 \cup \Phi$ can be closed with equality
            then success
            else fail
     else   if switched on, try to dissolve;
            choose a formula $\phi$;
            apply rule to $\phi$;
            reorder conclusion $\bar{C}$;
            call close_multiple recursively with $B \cup E_1 \cup \Phi$ and $\bar{C}$
     fi
fi;
call close_multiple recursively with $B$ and $E_2, \ldots E_n$.
```
■

The initial tableau is treated as an empty branch with a conclusion that consists of a single extension, namely the initial branch. Note that not all formulae in the current knowledge base

[1] The if-then-else constructs represent *choice points*, i.e., the else-part of a construct is not only executed if the condition evaluates to false, but also if the then-part fails.

need to reside on the initial branch, rather they are fetched on demand during the proof. This is advantageous when the knowledge base is large, but loosely connected.

From the algorithm shown above it is clear that several tasks can be separated. The predicate `close_multiple` itself resides in the module `main`. Immediate closure between atomic formulae is checked in `closure`, possibly using `unification` if sorted terms occur. Since there may be several candidate pairs for a closure, `heuristics` is needed. The `heuristics` module is needed in the next step as well to determine which formula to retrieve next from the current knowledge base. It uses static indexing information for this. Since the knowledge base may be empty or contain merely atomic formulae a branch can become exhausted at this point. If the problem contained equalities, the branch is tried to be closed with the equality theory available at this point. If the branch is neither closed nor exhausted the expansion process is continued. The module `choice` selects the next formula in focus, while `inference` contains the predicates that perform a rule application. Obviously, the information from `rules` is needed here. Another predicate from `inference` reorders the newly generated conclusion to achieve fairness.

We emphasize that all relevant data structures are encapsulated in the module `datastructures` and can only be accessed through the interface predicates of that module. This information hiding technique proved to be helpful several times when the internal representation underwent refinements and changes. Similarly, all output predicates and error message texts are collected in separate modules.

Unification with occur check is not provided as a built-in in Quintus Prolog resp. SICStus Prolog and thus is programmed in Prolog. An attempt to gain a speed-up by using a C implementation proved to be in vain. The reason was the very rudimentary C interface of Quintus Prolog (Gerberding, 1990).

The modules which together form the *core* of $_3T^AP$ are thus:

`choice`	`complete`	`equality`	`inference`
`rules`	`closure`	`dissolve`	`heuristics`
`main`	`unification`		

The following modules provide general services or have a technical nature:

`datastructures`	`msg_tap`	`sysdep`	`globalvars_quintus.c`
`declarations`	`output`	`globalvars.c`	`globalvars_sicstus.c`

These are the modules which form the *shell*:

`boot`	`index`	`information`	`interface`
`makekbx`	`preproc`	`proveall`	

The following modules form the compiler:

`scanner.l`	`grammar.y`	`output.c`	`output.h`

The following table summarizes $_3T^AP$'s modules:

Module Name	Short Description
`boot`	Loads and compiles all required modules; initializes various variables.
`choice`	Predicates for choosing the next formula in focus on the current branch.
`closure`	Predicates to check the current branch for closure with the current extension.
`complete`	Predicates for implementing completion-based equality reasoning.
`datastructures`	Predicates that implement the main data structures. All main data structures can solely be accessed via this module.
`declarations`	Global declarations concerning defaults for switches, initial signing, file names, operator names etc.
`dissolve`	Predicates for implementing the dissolution rule.
`equality`	Predicates for implementing equality reasoning.
`heuristics`	Representation of heuristics for selection of closure pairs and next formula to retrieve from knowledge base.
`index`	Generation of static indexing information stored in compiled knowledge bases.
`inference`	Application of tableau rules and construction of conclusion in a fair order from information in the `rules` module.
`information`	Predicates and texts of on-line help.
`interface`	Provides top-level predicates for compiling knowledge bases, doing proofs in various manners etc. Calls the compiler.
`main`	Provides the predicate `close_multiple` which implements the main loop of the prover.
`makekbx`	Predicates for handling the compiler output and adding index and control information and for loading a thus preprocessed KB into the workspace. Adds index and control information.
`msg_tap`	Contains error message texts. Not a module in the sense of Quintus (resp. SICStus) Prolog.
`output`	Predicates for writing formatted output information to streams. All output to the user is handled via this module.
`preproc`	Predicates for preprocessing formulae.
`proveall`	Predicates for testing a complete suite of problems and generating statistics.
`rules`	Representation of the tableau rules.
`sysdep`	Contains all predicates that are not conforming to DEC-10 Prolog, Unix specific predicates and C interface predicates.
`unification`	Predicates for unification of sorted terms.
`globalvars.c`	C program for global variable management.
`globalvars_quintus.c`	C program which is needed in addition to `globalvars.c` if Quintus Prolog is used.
`globalvars_sicstus.c`	C program which is needed in addition to `globalvars.c` if SICStus Prolog is used.
`scanner.l`	Contains the rules for the syntax-analysis. This file is the input for the Unix tool Lex (or Flex) to generate the scanner.

Module Name	Short Description
`grammar.y`	Contains the grammar rules. This file is the input for the Unix tool Yacc (or Bison) to generate the parser.
`output.c`	Contains the output procedures for the parser. This file is written in C.
`output.h`	Header-File for `output.c`.

We give the call dependency graph for the modules of the core and the shell in Figure 4.1. For sake of simplicity we did not include `dissolve` which is called only optionally and `heuristics` which is called by a number of modules. Finally, Table 4.1 gives some statistics of $_3T^AP$'s source code (incl. comments).

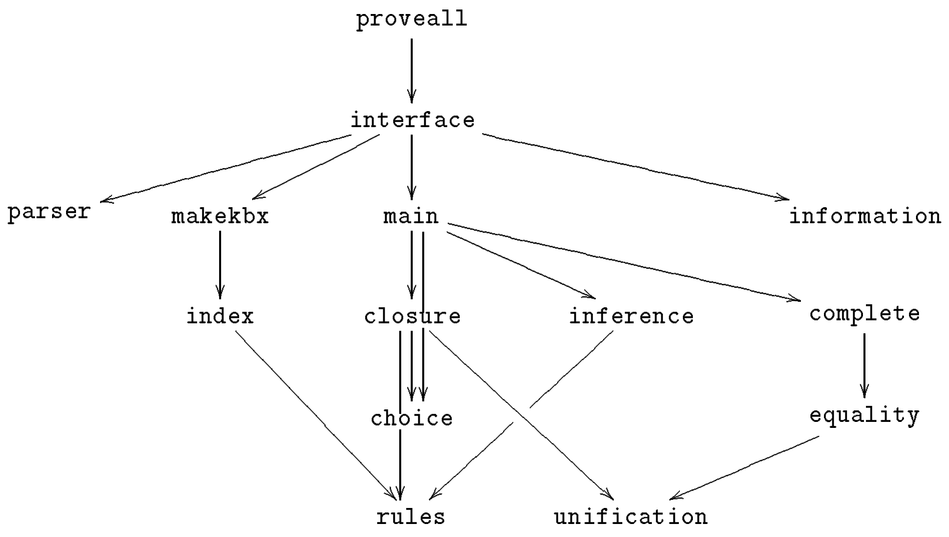

Figure 4.1: Dependency graph for the modules of $_3T^AP$'s core.

	Language	No. of lines	KByte
Prover	Prolog	23,600	799
	C	1,200	30
Compiler	C	1,600	46
	Yacc/Bison	500	14
	Lex/Flex	200	5
Utilities	C	2,400	89
	Lex/Flex	400	11
Total		29,900	994

Table 4.1: Statistics of $_3T^AP$'s source code (incl. comments).

5 System Description by Modules

5.1 Proveall, Information, Boot, Interface

5.1.1 The User Interface

The descriptions of these modules are collected in one section because these four modules together provide the user interface for the prover $_3T^AP$.

With the module `boot`, the whole system is compiled and loaded into the workspace of Prolog. `information` supports several help pages and `proveall` can be used to compile and prove whole sets of problems. `interface` contains the predicates for starting a proof. The following four sections describe each module in detail.

5.1.2 Proveall

This module provides predicates `proveall/1,2,3` and `compall/1`; they have the following form:

- `proveall( +What_to_prove )`
- `proveall( +What_to_prove,+Format )`
- `proveall( +What_to_prove,+Parameter )`
- `proveall( +What_to_prove,+Format,+Parameter )`
- `compall( +What_to_prove )`

With the predicates `proveall/1,2,3`, sets of problems can be proved, and a file `statistics` can be generated containing statistics on the proof length, time etc. With the predicate `compall/1`, sets of problems can be compiled.

The predefined problems are subdivided into groups as shown in Table 5.1. For a detailed description, see Section 8.1.

tests	dagostino	mr
cr	meta_pl	pigeon
pig_alt	kalish	ps
groups	pel_prop	pel_pred
pel_eq	phi	three_valued

Table 5.1: Groups of test problems for $_3T^AP$.

The first argument of `proveall/1,2,3` and `compall/1` can be either

- the name of a problem set,
- a list of problem set names,
- the keyword `pelletier` to prove or compile the three sets `pel_prop`, `pel_pred` and `pel_eq`, or
- the keyword `all` to prove or compile all the problem sets.

Example 5.1 *Some examples for the usage of* `proveall/1`*:*

- `proveall( pel_prop ).`
  proves the set `pel_prop` *which contains 17 propositional problems from* (Pelletier, 1986)*.*
- `proveall( [pel_prop,pel_pred] ).`
  proves the sets `pel_prop` *and* `pel_pred` *which contain 17 propositional and 28 first-order problems from* (Pelletier, 1986)*.*
- `proveall( pelletier ).`
  proves the sets `pel_prop`*,* `pel_pred` *and* `pel_eq` *which contain 62 problems from* (Pelletier, 1986)*.*

Remark 5.2 *As indicated by its name, the problem set* `three_valued` *can be proved (resp. compiled) only with the three-valued version of $_3T^AP$. In the two-valued version,* `all` *does not include the problem set* `three_valued`*.*

If `proveall/1` is used or anything but the keyword `tex` is used as the second argument of `proveall/2` or the third argument of `proveall/3` the statistics file is written as a standard text file, else the statistical informations are formatted such that they can be included into a LaTeX file.

Example 5.3 *Some examples for the usage of* `proveall/1` *and* `proveall/2` *with the formatting option:*

- `proveall( all ).`
  proves all problem sets and writes the statistic information as a text file.
- `proveall( all,ascii ).`
  does exactly the same.
- `proveall( all,tex ).`
  proves all problem sets and generates a LaTeX file.

The problems are proved using the command `proveinc` (but note the exception in Remark 5.5), i.e., either the parameter *maxcounter* or the parameter *maxbranchlength* is increased until a proof is found. By default *maxcounter* is used. *maxbranchlength* is increased if one of the keywords `maxbranchlength` and `mbr` is used as the second argument of `proveall/2,3`. The command `proveinc` is described in Section 5.1.5.

Each problem in a problem set is written in form of a list containing three members. The first member is the name of the problem, the second member is a list containing the theorems to be proved and the third member is also a list, containing the setting of the switches that should be used. If this list is empty, the proof is tried with the default settings.

Example 5.4 *Two problems from the problem set* `pel_pred`*:*

- `[pel23,[pel23],[]]` *means: Prove the theorem* `pel23` *from the problem* `pel23` *and use the default settings.*
- `[pel34,[pel34],[set_maxcounter(1)]]` *means: Prove the theorem* `pel34` *from the problem* `pel34` *and set* maxcounter *to 1.*

Remark 5.5 *If a command* `set_maxcounter(`n`)` *(resp.* `set_maxbranchlength(`n`)`*) is contained in the list of settings associated with a problem, and* maxcounter *(resp.* maxbranchlength*) is the parameter to be increased, the problem is proved by calling* `prove` *using the value n for* maxcounter *(resp.* maxbranchlength*) instead of using* `proveinc`*.*

5.1.3 Information

The `information` module is a module providing on-line help. It gives information about the available commands.

It provides two predicates, `info/0` and `info/1`. If you call `info/0`, you get an overview over the available information pages concerning the following topics: compiler, prover, equality, diss, (dissolution), maintain (maintaining the workspace), variables, output (tableau output), unix, info, all.

With the call

```
info( +What_information )
```

where `What_information` is one of the keywords listed above, you can get further information concerning these topics, that is, hints for the setting of global variables, the current settings, a description of the equality strategy or an overview over the available unix commands etc.

Example 5.6 `info.` *shows a list of the available information pages.*

`info( prover ).` *shows a list of available commands for proving a problem.*

`info( all ).` *prints all available information pages. It is recommended to select an output stream other than the screen if you want to read all information. See also Section 5.14.*

5.1.4 Boot

The `boot` module is the access to $_3T^AP$. After invoking Prolog, you have to type

```
| ?- compile( boot ).
```

to build the $_3T^AP$ system. The following actions take place:

1. Compile the modules.
2. Initialize the prover.
3. Initialize the settings.
4. Print the information overview page.

After that, the whole user interface can be used in order to work with $_3T^AP$.

5.1.5 Interface

The module `interface` contains predicates to compile, read, check and delete knowledge bases as well as to start the prover. The predicates can be classified as follows:

5.1.5.1 Predicates Concerning Knowledge Bases

The following predicates are used to create, manipulate, use and delete a knowledge base (KB) and extract information about a selected KB:

- `compkbx( +File ).`
  Calls the compiler for parsing the formulae in `File` and generates the static index. As the result of the call, a file `file.kbx` is created and written to the current working directory. This file is in the format direct input as a KB.
- `readkbx( +File )`
  Use a `*.kbx` file and read it as a KB into the workspace.
- `usekbx( +File )`
  Compile a file and load it as a KB into the workspace. The effect is the same as executing first `compkbx( File )` and then `readkbx( File )` for some file.
- `delkbs`
  Delete all KBs from the workspace.
- `delkb`
  `delkb( +KB )`
  Delete the specified (or by default the current) KB from the workspace. If afterwards the workspace is not empty, the current KB is set to an arbitrary one available.
- `writekb`
  `writekb( +KB )`
  Write all formulae of the specified (or by default the current) KB to the current output stream.
- `writeidx`
  `writeidx( +KB )`
  Write all indexing information of the specified (or by default the current) KB to the current output stream.
- `writesort`
  `writesort( +KB )`
  Write all information on sorts of the specified (or by default the current) KB to the current output stream.
- `writekbx`
  `writekbx( +KB )`
  Write all formulae, index entries and sorts of the specified (or by default the current) KB to the current output stream.

Example 5.7 *Suppose we have a file* `test` *that contains a problem to be proved.*

- `usekbx( test ).`
  compiles the file `test` *, creates a file* `test.kbx` *and loads* `test.kbx` *into the workspace of Prolog. A list with theorems available is sent to the current output stream.*

- `readkbx( test ).` *and*
  `compkbx( test ).`
  in combination have exactly the same effect.

- `delkb( test ).`
  removes `test.kbx` *from the workspace of Prolog. If you have only one* `*.kbx` *file in the workspace or* `test.kbx` *is the current KB,* `delkb.` *has the same effect.*

5.1.5.2 Predicates Concerning the Proof

The following predicates are used to prove something from a KB:

- `prove`
  `prove( +Theorem_index )`
  `prove( +Theorem_index,+KB )`
  Call the prover to prove the theorem indicated by `Theorem_index`. The default KB is the current one and if no theorem index is specified, it is assumed that the current KB is checked for consistency.

- `proveinc`
  `proveinc( +Theorem_index )`
  `proveinc( +Parameter )`
  `proveinc( +Theorem_index,+KB )`
  `proveinc( +Parameter,+Init )`
  `proveinc( +Theorem_index,+Parameter,+Init )`
  `proveinc( +Theorem_index,+KB,+Parameter )`
  `proveinc( +Theorem_index,+KB,+Parameter,+Init )`
  Call the prover to prove the theorem indicated by `Theorem_index`. The default KB is the current one and if no theorem index is specified, it is assumed that the current KB is checked for consistency. Either the parameter *maxcounter* (default) or the parameter *maxbranchlength* (if `Parameter` is `maxbranchlength` or `mbr`) is increased until a proof is found or a limit is reached. `Init` is the initial value for the parameter to be increased. If no value is given, 0 is used for *maxcounter* and 1 for *maxbranchlength.* The limit is given by the parameters *inc_limit_mc* and *inc_limit_mbr.*

- `proveinc_return_mc( +Parameter,-Value )`
  `proveinc_return_mc( +Theorem_index,+Parameter,-Value )`
  These predicates do exactly the same as `proveinc/1` and `proveinc/2`. In addition they give back the smallest *maxcounter* (resp. *maxbranchlength* sufficient for the proof.

- `protprove`
  `protprove( +Theorem_index )`
  `protprove( +Theorem_index,+KB )`
  Prove a theorem from a specified (or by default the current) KB and protocol the proof in a previously specified output file. With `protprove/0`, a consistency check is tried.

- `inconsistent`
  `inconsistent( +KB )`
  These predicates try to close a tableau initially consisting of only the axioms contained in KB, i.e. they try to prove the inconsistency of a KB.

Example 5.8 *Let us continue the example from above. Suppose is a theorem* `th_test` *in the knowledge base. Then*

- `prove( th_test ).` *and*
  `prove( th_test,test ).`
  do the same, namely to prove the theorem `th_test` *using the default parameter* maxcounter.

- `proveinc( th_test ).` *and*
  `proveinc( th_test,test ).`
  also do the same, that is, they prove `th_test`*, starting with* maxcounter = 0*, increment* maxcounter *every time the proof fails and start the whole proof procedure again with a higher value.*

- *Respectively,*
  `prove.` *and*
  `proveinc.`
  try to prove the inconsistency of the axioms of `test`*.*

- `inconsistent.` *and*
  `inconsistent( test ).`
  do exactly the same as `prove` *and* `prove/1`*.*

5.1.5.3 Help and Information Predicates

There are two more predicates exported by this module:

- `lookup`
  This predicate shows the current settings of all switches and parameters.

- `stepcontrol`
  Checks whether `step_mode` is `on` and if so waits for a command to be entered before continuing. Possible input commands are:
  - `c` : continue
  - `a` : abort
  - `h` : help
  - `l` : set step mode off
  - `d` : set the debug level to 0
  - `e` : set the equality debug level to 0
  - All other commands print the information page concerning step modes.

5.1.5.4 How Does It Work?

The predicates concerning the proof differ in various settings and in the proof strategy, but have in common that they all call the central predicate of the `interface` module which is the predicate

```
tap( +KB,+Theorem_index,+Idcs )
```

This predicate calls $_3T^AP$ to prove the theorem with the index indicated by `Theorem_index` from the denoted KB and ignore those which are listed in `Idcs`. The predicate works as follows:

1. Set and reset global variables.

2. Select the desired output stream.
3. Initialize the branch according to the desired proof, i.e. put the theorem on the branch if this was desired and put further axioms on the branch according to the setting of the switch *grepall*. *grepall* controls the initialization of the branch as follows:
   - If *grepall* is on (the default), then all formulae from the workspace which have an atomic link to the theorem are "grepped" onto the branch. This is done using the predicate `add_th_connected_fmae_to_branch/5`.
   - If *grepall* is off, all atoms (and only atoms) from the workspace which have an atomic link to the theorem are "grepped" onto the branch. This is done using the predicate `add_th_connected_atom_to_branch/5`.
4. If no theorem is to be proved, an arbitrary axiom is "grepped" from the workspace and is used for the initialization of the branch.
5. Then it is checked if you actually use sort information for the proof (i.e. if you use more than one sort).
6. Finally, prove it: this is done via the `close_multiple/4` predicate in the `main` module, for further details see Section 5.2.
7. When the proof search terminates, statistics are print and a message that tells whether a proof has been found.

5.2 Main

This module contains the central predicate

- `close_branch/3`

This is the implementation of the basic algorithm of (Hähnle, 1990a) for a multi-valued automated theorem prover based on the tableau method.

`close_branch/3` works as as follows:

First a signed formula (sformula) is selected via the `choose_sformula/4` predicate from the `choice` module, which is described in Section 5.4.1.

Due to the possibility that `choose_sformula/4` could result in grepping some new sformulae from the workspace on the branch, we have to examine immediately whether the branch can be closed with the selected sformula (treated as an extension). In that case, nothing else is done and the predicate succeeds. Otherwise, there are several possibilities:

1. The flag `dissolution` is set to `on`. Then it is tried to close the branch via dissolution. If this succeeds, nothing else is done and `close_branch/3` succeeds.
   If `dissolution = off` or no closure is possible with dissolution, then there are three more possibilities:
2. The branch is not exhausted. This is the most common case. The branch then is expanded via the `apply_rules/4` predicate in the `inference` module and the resulting new branch(es) is (are) tried to be closed by the `close_multiple/4` predicate, which represents the transitive closure of `close_branch/3` (see below).

3. The branch is exhausted and it is possible to fetch new formulae from the workspace. One new formula is put onto the branch and `close_branch/3` is called recursively.

4. The branch is exhausted and there is no formula in the workspace that can be put on the branch:

   - If the flag `equality` is set to `on`, the branch is tried to be closed via equality. This is done by the predicate `close_branch_with_completion/1` from the `complete` module, see Section 5.13.
   - In all other cases, the predicate `handle_exhausted_branch/1` is called and `close_branch/3` fails. Prolog then tries to find a proof with backtracking.

The other predicate of the `main` module,

- `close_multiple/4`

is both the entry into the whole proof procedure as well as the controlling predicate for the closure of the whole tableau. `close_multiple/4` tries to close the current branch together with the first extension of the current conclusion.

- This conclusion is in the initial step created by the `tap/3` predicate from the `interface` module and is either the theorem to be proven or an arbitrary formula in the case of an inconsistency proof.
- Afterwards the current conclusion is the result of the `apply_rules/4` predicate applied on a selected sformula of the current branch.

If it is possible to close the branch with the first extension, `close_multiple/4` is called recursively with the remaining extensions of the current conclusion, otherwise `close_branch/3` is called with the first extension added to the branch. If the branch—together with one of the extensions—gets longer than *maxbranchlength* (cf. Appendix app-switches), `close_multiple` fails.

Interaction schema:

The predicate `close_multiple/4` has the following structure:

- Entry:
```
close_multiple( Branch,Conclusion,Diss_info,No ) :-
                               ...
     check_closure( +Branch,_,+First_extension,_,_ ),...
```

- If check_closure/5 fails
```
     close_branch( New_branch,Diss_info,No ),
                               ...
     close_multiple( New_branch,Remaining_extensions,Diss_info,No ),...
```

The predicate `close_branch/3` has the following structure:

- Entry:
```
close_branch( Branch,Diss_info,No ) :-
     choose_sformula( Branch,Sformula,Temporary_branch,Added_formula ),
     check_closure( Branch,_,First_extension,_,_ ),...
```

- If `check_closure/5` fails, but `choose_sformula/4` has found a new sformula:
  `treat_dissolution/10`
- If the branch is still not closed
  `apply_rules/4,`
  `close_multiple/3,...`
- If no new sformula has been found to expand but there are still unused sformulae in the workspace, then grep one new formula, put it on the branch and call
  `close_branch/3`
- If no new expandable sformula has been found and no more sformulae are in the workspace and the `equality` flag is set:
  `close_branch_with_completion/1`
- Otherwise:
  `handle_exhausted_branch/1,`
  `fail.`

5.3 Closure, Heuristics

We describe these two modules in the same section, because the **`heuristics`** module contains the predicate `select_complementary_atoms/2` that represents the heuristic according to which a branch will be closed, in particular, when there are two or more possibilities for a closure.

5.3.1 Heuristics

This module contains the predicates for representation of the heuristics for proof search used in $_3T^AP$. These are heuristics concerning the closure of a branch as well as the selection of sformulae or atoms from the KBs for putting them on a branch.

5.3.1.1 Heuristic for Closure

The central predicate is:

- `select_complementary_atoms( +Complementary_atoms_list,-Selected_pair )`

`Complementary_atoms_list` is a list of pairs of atoms, which have complementary[1] signs (according to the corresponding predicate in the `declaration` module) and can be unified. This list is generated in the `closure` module, see Section 5.3.2.

The predicate selects the "best" complementary pair of all potential pairs. Therefore, two help predicates are needed:

- `construct_heuristic_list( +Complementary_atoms_list,-Heuristics_list )`
  This predicate orders the potential pairs according to the following heuristic:
  - Pairs whose unification does not lead to variable instantiations are selected primary.

[1] In the many-valued case signs correspond to sets of truth values and these are thought to be complementary iff they are inconsistent iff they have an empty intersection.

  - The secondary ordering is by the number of former usages in closures (to assure fairness).
  - Finally pairs are ordered by the number of variable instantiations necessary to unify them.

  Concisely: We choose the pair which does not lead to variable instantiations. If there is no such pair, we use the pair with the atoms which have been used fewest for closure so far. If this is not unequivocal we order the pairs according to the least instantiations of variables necessary by the unification of the atoms.

  A pair (a_1, a_2) is treated as if it had been used to close a branch B if B has been closed using atoms (a_1', a_2') and

  1. a_1 and a_1' are identical up to variable renaming,
  2. a_2' is an instance of a_2.

- `construct_pair_list( +Heuristic_list,+Acc,-Pair_list )`
  This predicate transforms the heuristic list from the previous predicate into a list of pairs of atoms by stripping off the additional heuristic information.

Via the `sysdep_member/2` predicate, we now select a pair of atoms and use this pair for the closure of the branch. This is in the first case the first pair of the list, but it represents also a choice point, and if backtracking does occur we run all over the list.

5.3.1.2 Heuristic for Sformula Selection

There are several predicates which control which sformula or atom to fetch from the KB. These are:

- `grep_connected_formula( +Sign,+Atfma,+KB,+Idcs,-Signed_fma,-Idx )`
  This predicate is used for fetching a signed formula (and its index) from KB which contains the atomic formula `Atfma` with a complementary polarity as indicated by `Sign`, but only if its index `Idx` is not yet contained in the indices list `Idcs` of the branch. (For additional information on indexing and related topics see Section 5.12). If there is no new sformula in KB connected to `Sign` and `Atfma` this predicate fails.

- `grep_atom( +Sign,+Atfma,+KB,+Idcs,-Signed_fma,-Idx )`
  Fetch an atom (and its index `Idx`) from KB but only if `Idx` is not yet contained in `Idcs`.

  Note: The first two arguments `Sign` and `Atfma` are not used in this version of $_3T^AP$.

- `grep_unconnected_formula( +Idcs,+KB,-Signed_fma,-Idx )`
  Returns a signed formula not on the current branch so far (as indicated by `Idcs` and its index `Idx`. If there is no such formula in KB, this predicate fails.

- `grep_formulae( +Branch,+Branch_no,-New_branch,-Closed )`
  This predicate adds formulae from the workspace to the branch before a rule is applied. All cases in which this has to be done are combined in this predicate. The cases are:

  1. It is possible to grep a formula by a new atomic link
     (a) It is possible to close the branch with the new formula. Nothing else has to be done, the flag `Closed` is set to `true`.

   (b) It is not possible to close the branch with the new formula and the new formula is an atom. The predicate calls recursively itself. This recursion is continued as long as it is possible to grep atoms by new atomic links and as long as these new atoms do not close the branch.

   (c) The new formula is not an atom. Nothing else is done, the flag `Closed` is set to `fail`.

2. There is no new formula on the branch and there are still new formulae in the workspace. One new formula from the workspace is added to the branch, `Closed` indicates, whether the new formula closes the branch.
3. In all other cases nothing is done and the flag `Closed` is set to `fail`.

Note: The predicate never fails.

- `new_grep_formulae( +Branch,-New_branch,-Added_formulae )`
  This predicate adds formulae from the workspace to the branch before a rule is applied. This is done only if no unused sformulae are on the branch. All cases in which this has to be done are combined in this predicate. The cases are:

  1. It is possible to grep a formula by a new atomic link.
     (a) The new formula is an atom. The predicate then calls itself recursively. This recursion is continued as long as it is possible to grep atoms by new atomic links.
     (b) The new formula is not an atom. Nothing else is done.
  2. There are new formulae on the branch and there are still new formulae in the workspace. One new formula from the workspace is added to the branch.
  3. In all other cases nothing is done.

The last two predicates both use a predicate `grep_new_formula/3` which is only used within the `heuristics` module.

- `grep_new_formula( +Branch,-New_Formula,-New_Index )`
  Get some formula from the workspace unused on the branch so far. If a theorem is to be proved the choice is restricted to a connected formula; if consistency is to be proven also an unconnected formula may be chosen.

  Note: The formula returned is ready to be added to the list of new formulae on the branch.

As mentioned above, the formulae grepped preferably from the workspace are the ones that have a new atomic link. This is tried via the following predicate:

- `grep_formula_by_new_atomic_link( +Branch,-New_Branch,-New_sformula )`
  If there is an unused atom on the branch with a link to a formula not yet marked, this formula is grepped from the workspace and added to the branch. If there is no such formula, the predicate fails.

Finally, there are two predicates which handle grepping of formulae in connection with the equality strategy (See Section 5.13 for more details).

- `grep_formula_for_exhausted_branch( +Branch,-New_branch )`
  This predicate adds a formula from the workspace to an exhausted branch. In this version, a formula that contains an equality is added. If there are no such formulae in the workspace or the flag `equality` is off, the predicate fails.

- `grep_equality_without_link( +Branch,-New_Branch )`
  If there is a formula in the workspace that has a link to an inequality or a negated demodulator, it is added to the branch.

The predicates stated so far control grepping of formulae and atoms on the current branch. They are mainly used during initialization of a branch and in the case when there are no more expandable sformulae on a branch.

There is one more (help) predicate in this module:

- `get_free_vars( +Fma,-Var_list )`
  Gives back a list of the free variables in the formula `Fma`.

5.3.2 Closure

This module contains the predicates which check a branch together with an extension for a possible closure. The central predicate is:

- `check_closure( +Branch,+Branch_no,+Extension,-Updated_branch,-Flag )`

It works as follows:

- `check_closure/5` is called. If `Branch` together with `Extension` can be closed `Flag` is set to 1. `Updated_branch` contains the modified branch, that is, some variables may have been instantiated and the counter of an atom may have been increased. The argument `Branch_no` is used for statistical and output purposes.

  In order to check for a closure,

- `close_extension( +Branch,+Branch_No,+Extension,-Close_flag,-New_branch )`
  is the first predicate called. It tries to close `Branch` together with `Extension`. If a closure is possible, `New_branch` is the updated version of `Branch`.

  The separation between `check_closure/5` and `close_extension/5` has historical reasons. In an older version of $_3T^AP$ we examined all extensions we got back from a rule application for a possible closure with `Branch`. Therefore, `check_closure/5` was the control predicate and worked with a list of extensions rather than only one. `close_extension/5` was called by it for each of these extensions.

  `close_extension/5` uses the following predicates:

-
  ```
  collect_atoms_of_extension( +Extension,+Atm_list,-Close_flag,
                              Branch,Branch_No )
  ```
  This predicate has two functions: First, it collects all atoms of the extension and second, if during the search for atoms a sformula is found, it is checked here, whether a tableau rule is defined for it. In the two-valued version, this is always the case, but in the three-valued version, e.g. for `[_,_,_,wneg(Fma),uSign]` no rule is defined and the branch in which it does occur is closed.

  If we have found atoms or the branch is not already closed,

- `search_complementary_atoms_of_extension( +Atm_list,-Compl_atoms_list )`
  is called. It checks, whether the extension itself contains complementary atoms. If we have found some,

- `add_flag_to_pairs( +Pair_list,ext,-Triple_list )`
  is called to record that this complementary pair was found in the extension (`ext` is the keyword for extension).

  Next, via `get_all_atoms_of_branch/2` from the `datastructures` module and the predicate
- `search_complementary_atoms( +Ext_atms_list,+Branch_atms_list,`
  `-Comp_atms_list )`
  we check, whether there are atoms on the branch being complementary to the atom(s) in the extension. Again we record this fact via
- `add_flag_to_pairs( +Pair_list,branch,-Triple_list )`.

If we have found complementary atoms, we use the predicate `select_complementary_atoms/2` from the `heuristics` module to select the best one.

Now we do some statistics, instantiate the necessary variables by unification and eventually update the branch and set `Close_flag`. Some help predicates of `close_extension/5` are explained now:

- `check_if_no_rule_defined( +Branch,+Sfma,-Close_flag,-New_branch )`
  This predicate checks if a closure of the branch is possible because there is no rule defined for the sformula `Sfma`, see above.
- `search_complementary_pairs( +Atom,+Atom_list,-Pair_list )`
  This predicate looks, whether in `Atom_list` one or more atoms exist which are complementary to `Atom`. Each complementary pair is added to `Pair_list`. If `Atom` is an equality or a demodulator and equality is switched on the results is the empty list.

Two more predicates are in `closure` are related to closure checking:

- `check_closure_by_sformula_from_kb( +Branch,+No,+New_sfm,`
  `-New_branch,-Closed )`
  Checks, whether the branch can be closed with the help of `New_sfm`, which is taken from the workspace. This predicate is used in association with the `grep_*` predicates from the `heuristics` module and simply interprets `New_sfm` as an extension and then calls `check_closure/5`.
- `handle_exhausted_branch( +Branch )`
  This predicate does everything what has to be done when a branch is exhausted, in particular the updating of statistical information. It is normally called before backtracking starts.

Finally,

- `substitute_univ_vars_sorted( +Atm,+Univ_var_list,`
  `-New_atm,-New_univ_var_list )`
  is included in this module which substitutes the variables in `Atm` with respect to which it is universal by new ones which have the same sort as the old ones.

5.4 Choice, Inference

During the proof procedure the predicate `choose_sformula/4` in the `choice` module usually supplies the next sformula to be expanded via `apply_rules/4` in the `inference` module. A detailed description of these two modules is given in the following two sections.

5.4.1 Choice

This module contains predicates for the selection of sformulae. During a proof, two kinds of sformulae have to be chosen from the branch:

1. If the branch has to be expanded, we need a sformula to serve as premise for a tableau rule application.
2. If we look for a possible closure of a branch, we need an atom from the branch which is able, together with an atom from the current extension, to close the branch.

The two associated predicates are

1. `choose_sformula( +Branch,-Sformula,-New_branch,-Added_formulae )`
   This predicate selects the next sformula to be expanded. If the chosen sformula

   - is not linked to another sformula on the branch or
   - is not linked to an axiom in the workspace and or
   - does not contain an (in)equality.

   and *removeunlinked* is switched on, it is removed and another sformula is chosen.

   The selection of the sformula proceeds in the following precedence:

   (a) Look for an unused[2] sformula on the branch. If there is no such sformula,
   (b) look for an unused sformula in the workspace. If there is no such sformula,
   (c) look for an already used sformula from the branch (this must be a γ-formula!). If there is no such sformula,
   (d) no sformula has been found to be expanded next.

   `New_branch` can contain more or fewer sformulae than `Branch`. The list `Added_formulae` contains the formulae added to the branch.

   Note: `choose_sformula/4` never fails. If no sformula can be chosen, the atom `none` is returned as value of the Prolog variable `Sformula`.

2. `choose_atom( +Branch,-Atomic_fma )`
   This predicate selects an atom from the branch. First it looks for atoms which have not been used for a closure, and if all unused atoms are examined, the already used atoms are selected. This is done via the `get_best_atom_of_branch/2` predicate in the `datastructures` module.

The other predicates in this module are used to judge the usefulness of the chosen sformula. These are:

- `can_be_removed_from_branch( +Sformula,+Branch )`
  Succeeds, if `Sformula` can be removed from the branch, i.e.

  – it is not linked to a formula on the branch or
  – it is not linked to an axiom in the workspace or
  – it does not contain an equality (only if `equality` is set to `on`).

[2] That is, it has not been used before as a premise for rule application.

- `contains_an_equality( +Sformula,+KB )`
  Succeeds, if `Sformula` contains an equality or an inequality. We need the current KB, because this information has already been computed and is stored in the associated lookup table of KB.

- `is_linked_to_branch( +Sformula,+Branch,+KB )`
  Succeeds, if `Sformula` is linked to one of the sformulae on `Branch`.

- `is_linked_to_axiom_not_on_branch( +Sformula,+Branch,+KB )`
  Succeeds, if `Sformula` is linked to one of the axioms in the workspace not on the branch yet.

- `are_linked_sformulae( +Sformula_1,+Sformula_2,+KB )`
  Succeeds, if the two sformulae are linked.

5.4.2 Inference

This module contains the predicates for the application of tableau rules and the construction of appropriate conclusions.

The main predicate is

```
apply_rules( +Branch,+Sformula,+Maxcounter,
             -Conclusion,-Updated_branch )
```

It applies the proper tableau rule (fetched via the `rule/8` predicate of the `rules` module) to `Sformula`. For further details regarding rule application, see Section 5.10.

The conclusion is constructed using the appropriate predicates in the `datastructures` module. This conclusion is then rearranged via `juggle_conclusion/2`, which is used to achieve fairness. For fairness the labels associated with each operator in each sformula are used. With each label a global counter is associated (see Section 5.5).

Then, if the flag *uselemmata* is either `alpha` or `on`, the predicates `add_lemmata_alpha/8`, respectively `add_lemmata/8` are used to generate lemmata which are added to the conclusion. For a description of lemma generation, see Section 2.4, as well as 5.10 and 5.5.

5.5 Data Structures

All data structures used by the $_3T^AP$ system are defined in the module `datastructures`. Access to data structures is provided through the access predicates defined in that module. The data structures and the access predicates are discussed in the following subsections. The system dependent parts of the module `datastructures` are hidden in the module which is described in Subsection 5.6. Some of $_3T^AP$'s data structures need global variables; these are implemented using the C programming language and the foreign language interface of Quintus (resp. SICStus) Prolog. The semantics of the interface and the implementation of the global variables in the module `globalvars` are described in Subsection 5.7.

5.5.1 Representation of Terms

The basic data structures for manipulating tableaux are terms and formulae. The representation of the former is discussed in this subsection, the latter are handled in Section 5.5.2.

As stated in Chapter 3 every term in the $_3T^AP$-system has some specific sort. The sort information is specified as a list which determines the complete path from the root of the sort hierarchy[3] to that sort plus an additional variable tail. Consider the sort declaration from Example 3.2:

```
sort fields.
sort real < fields.
sort complex < fields.
sort rational < real.
```

The sort information for a term of sort `fields` is `[fields|_]`. The sort `rational` is represented by `[fields,real,rational|_]`. Please note the variable tail albeit `rational` does not have any subsort. The variable tails for the same sorts need not to be the same.

A term is a constant or variable symbol or a function application followed by a colon and the list which represents the term's sort. For example, in the scope of the above sort declaration and

```
constant a,b : real.
function f : real x real -> rational.
```

the term `f(a,b)` is represented by the following Prolog term (white space has been added to achieve readability):

```
f( a:[fields,real|_ ], b:[fields,real|_ ] ) : [fields,real,rational|_ ].
```

Here are the predicates supplied by the module `datastructures` to access the data structures associated with terms and sorts:

- `get_term_of_sorted_term/2`
  Determine the (unsorted) term of a sorted term, i.e. the part to the left of the colon.
- `get_sort_of_sorted_term/2`
  Determine the sort of a sorted term, i.e. the list to the right of the colon.
- `split_sorted_term/3`
  This predicate splits the sorted term (hence the name) into two parts: The (unsorted) term as returned by `get_term_of_sorted_term/2` and the list returned by `get_sort_of_sorted_term/2`.
- `get_simple_sort/2`
  As stated above, the sort information is represented by a list with an uninstantiated tail. The list represents the path from the root of the sort hierarchy to the sort which is to be represented. This predicate determines the last instantiated element of that list.
- `get_incomplete_sort/2`
  Returns the uninstantiated tail of the list described above.
- `get_simple_and_incomplete_sort/3`
  Does the same as `get_simple_sort/2` and `get_incomplete_sort/2` together.

[3] Which indeed is a tree.

5.5.2 Representation of Formulae

Formulae are represented by special Prolog terms. The form of these terms is discussed in the following subsections.

5.5.2.1 Labels

Every formula in the ${}_3T^AP$-system has a label attached to it. The label is used to identify this formula throughout the whole tableau construction and to index a special data structure as described in Section 5.7. In the current version of the ${}_3T^AP$-system labels are built from Prolog atoms starting with the character `l` followed by some integer (e.g. `l17`). New labels are generated by the `genlabel/1` predicate. The predicate `is_label/1` may be used to test a given atom for labelhood.

5.5.2.2 Atomic Formulae

The simplest formula is an atomic formula, which is a propositional variable or a predicate. Atomic formulae are represented by Prolog terms whose leading functor is `atfma/2` (which is an abbreviation of *atomic formula*). The term's first argument is the predicate or propositional variable and the second argument is a label. For example, the atomic formula `p(a,b)` is represented by the Prolog term `atfma( p( ...,... ), l1 )`. Terms have been omitted for ease of reading. `l1` is a label. The predicate `is_atomic_fma/1` may be used to test, whether a formula is atomic. To extract the formula part of an atomic formula from the Prolog term use `get_atom_of_formula/2`. That predicate simply returns the first argument of the `atfma/2` term.

5.5.2.3 Compound Formulae

Like atomic formulae compound formulae are represented by certain Prolog terms. There is a Prolog functor for every connective in the given logic. The arity of that functor is one plus the arity of the connective. The additional argument is used to attach a label to every subformula. The label is always the last argument of the functor. Table 5.2 lists the functors associated with the connectives of the two-valued version of ${}_3T^AP$ and table 5.3 does the same for the three-valued version. The connectives not used in the standard version of the ${}_3T^AP$ system are associated with functors of the same name, e.g. the functor for the connective ~> is `~>/3`.

connective	functor/arity
`v`	`dis/3`
`&`	`con/3`
`=>`	`imp/3`
`<=`	`pmi/3`
`<=>`	`equi/3`
`-`	`sneg/2`

Table 5.2: Two-Valued connectives and associated functors.

Quantified formulae are represented in the same way as simple compound formulae. The functor used for an existential quantified formula is `ex/3` and for an universal quantified formula `all/3`.

connective	functor/arity
v	dis/3
&	con/3
=>	imp/3
<=	pmi/3
<=>	equi/3
#	reg/3
-	sneg/2
~	wneg/2
nabla	nabla/2
jt	jt/2
aff	partaffirm/2
ju	ju/2
jf	jf/2

Table 5.3: Three-valued connectives and associated functors.

Example 5.9 *Consider the following declaration:*

```
sort top.
predicate p,q : top.
variable x : top.
```

The formula `forall x (p(x) v q(x))` *is represented by the Prolog term*

```
all( x:[top|_ ], dis( atfma(p(x:[top|_ ]),l1),
                      atfma(q(x:[top|_ ]),l2),l3),l4).
```

The variable declaration for `x` *is not necessary (cf. Section 3).*

The following list shows the access predicates for formulae.

- `mk_formula/2`
  This predicate constructs a formula from a list which consists of an operator and one or two subformulae and a label.
- `get_op_of_formula/2`
  Determine the leading connective or quantifier of the given formula.
- `get_fma1_of_formula/2`
  Determine the left subformula of the given compound formula.
- `get_fma2_of_formula/2`
  Determine the right subformula of the given compound formula.
- `get_var_of_formula/2`
  Return the variable list of the given quantified formula.
- `get_arity_of_formula/2`
  Return the arity of the leading operator of the given formula. Please note that the arity of the Prolog functor is relevant here.
- `get_label_of_formula/2`
  Return the top level label of the given formula.

5.5.3 Representation of Signed Formulae

Signed formulae are represented as Prolog terms with the leading functor `sformula/8`. The arguments of `sformula` are

1. a usage counter for the number of rule applications to the formula,
2. the branching factor of the formula if a rule is applied to it,
3. the number of extensions a rule application will yield,
4. the term representing the formula, cf. Section 5.5.2,
5. the sign,
6. an index which is used to access some indexing data structure,
7. the name of the knowledge base it stems from,
8. the list of variables with respect to which the formula is universal.

Remark 5.10 *Please note that the branching factor and the number of extensions may be uninstantiated. These arguments are instantiated when the formula is considered first for a rule application. If an instantiation is necessary the values may be obtained by the* `rule/8` *predicate, cf. Section 5.10.*

Signs are represented as Prolog atoms whose names end in "sign" (e.g., the sign for "false" is represented by `fSign` and that for "unknown or true" is written as `utSign`).

The following predicates may be used to access the various parts of this data structure:

- `mk_sformula/9`
  This predicate builds a term consisting of the functor `sformula/8` and the first eight arguments. The term is returned using the ninth argument.
- `get_counter_of_sformula/2`
  Return the counter of the given signed formula.
- `get_branch_number_of_sformula/2`
  Return the branching factor of the given signed formula.
- `get_successor_number_of_sformula/2`
  Return the number of extensions generated by a rule application to the given signed formula.
- `get_sign_of_sformula/2`
  Determine the sign of the given signed formula.
- `get_fma_of_sformula/2`
  Extract the term representing the formula from the given signed formula.
- `get_kb_of_sformula/2`
  Return the name of the formula's knowledge base.
- `get_univ_vars_of_sformula/2`
  Return the list of variables with respect to which the given signed formula is universal.
- `increase_counter_of_sformula/2`
  Increase the counter associated with the given signed formula by one.

5.5.4 Representation of Extensions

An extension is represented as a Prolog list of signed formulae. The predicates below may be used to construct or access extensions:

- `mk_empty_extension/1`
  Return an empty extension, i.e. an extension containing no formulae.
- `get_sformula_of_extension/3`
  Returns the first formula from the extension and removes that formula from the extension.
- `add_sformula_to_extension/3`
  Adds the given signed formula to the extension. The result is the updated extension.

5.5.5 Representation of Conclusions

A conclusion is a collection of extensions. It is represented as a Prolog list of extensions, i.e. as a list of lists of signed formulae. Use the following predicates to build or access conclusions:

- `mk_empty_conclusion/1`
  Return an empty conclusion, i.e. a conclusion containing no extensions.
- `is_empty_conclusion/1`
  This predicate succeeds if and only if the conclusion is empty.
- `get_extension_of_conclusion/3`
  Extract the first extension from the conclusion and remove it from that conclusion.
- `add_extension_to_conclusion/3`
  Adds the given extension to the given conclusion.

5.5.6 Representation of Branches

In this section the representation of branches is explained. Not the whole tableau is constructed explicitly. Only the branch in focus is held in a special data structure which is described below. The information on the remaining structure of the tableau constructed so far is represented implicitly by the choice points of the Prolog system.

Branches are represented as Prolog terms with the principal functor `branch/6`. The argument positions of `branch/6` are used as follows (arguments from left to right):

1. The list of new[4] signed formulae on the branch. This is a Prolog list of terms. Every term represents a signed formula as described in Section 5.5.3. No rule has been applied to any of these formulae. The list does not contain any atomic formula.
2. The list of used signed formulae on the branch. This list is similar to the one above, but the formulae here have been used at least once for a rule application.
3. The list of new atoms on the branch. This list is similar to the list in Section 1, but the formulae here are atomic. No atom of this list has been used for branch closure.

[4] That is, no rule has been applied yet to that formula, cf. Section 5.5.3.

4. The list of used atoms on the branch. The atoms in this list have been used for branch closure at least once.
5. The list of indices of formulae on the branch, cf. Section 5.5.3.
6. The Dewey Number of the branch. The branch's Dewey Number is represented by a list of integers. An example is given below.

The lists in the branch data structure are sorted w.r.t. lexicographic ordering. This means that these lists of signed formulae are ordered by increasing usage counters.

Remark 5.11 *There is a problem with the lexicographic ordering. "10" is less than "2". This means if the γ-limit* maxcounter *exceeds ten then the ordering is no longer by increasing usage counters. In that case the* `get_best_sformula_of_branch/2` *described below will not return the "best" formula. This is a known* bug.

If two formulae have equal usage counters they are ordered w.r.t. their branching factor since the branching factor fills slot two of the data structure for signed formulae. The problem described above does not occur here because $_3T^AP$'s logics so far do not include rules with a branching factor greater than ten. For equal branching factors the ordering is w.r.t. the number of extensions a rule application to that formula will yield.

To access branches the following predicates are used:

- `mk_empty_branch/1`
  Return an empty branch, i.e. a branch without any formulae.
- `get_new_sformula_of_branch/2`
  This predicate returns an unused signed formula from the given branch. Like in the `member/2` predicate a different formula is chosen after each fail. The predicate fails if no formula is left.
- `get_used_sformula_of_branch/2`
  Similar to the previous predicate, this one returns a formula from the branch which has been used already.
- `get_best_sformula_of_branch/2`
  This predicate will return the best signed formula on the branch w.r.t. the following heuristic. An unused formula is better than a used one. If there are no unused formulae on the branch, the formula with the least usage counter is the best. This heuristic is implemented by fetching the first element of the list of new formulae (if any) or else the first element of the list of used formulae. There is a bug in this predicate, cf. Remark 5.11.
- `get_new_atom_of_branch/2`
  Return an unused signed atomic formula from the branch. Like the `member/2` predicate a different atomic formula is chosen after each fail. The predicate fails if no new atom is left.
- `get_used_atom_of_branch/2`
  Similar to the previous predicate, this one returns an atom from the branch which has been used already.
- `get_index_list_of_branch/2`
  Return the list of indices of formulae on the branch. Cf. item 5 in the above enumeration of the arguments.

- `add_index_to_index_list/3`
  Add an index to the index list of the branch. The result is the branch with the extended index list.
- `get_path/2`
  Return the Dewey Number of the branch.
- `add_sformula_to_branch/3`
  The predicate adds a signed formula (which may be atomic) to a branch. I.e. the formula is inserted in the appropriate list.
- `add_extension_to_branch/4`
  This predicate adds an extension to the given branch. The result is the updated branch which contains the signed formulae from the extension. The branch's Dewey Number is extended by the extension's number which is passed as an argument to this predicate[5].
- `update_branch/4`
  This predicate is called with a branch and a signed formula as arguments. The result is an updated version of the branch. There are three cases depending on the type of the formula, say Φ:
  1. If Φ is a γ-formula and its usage counter is equal to the γ-limit (`maxcounter`), it is removed. Otherwise Φ's counter is increased and Φ is possibly (if the counter was 0) moved from the list of unused formulae to the list of used formulae.
  2. If Φ is an atomic formula its counter is increased. If Φ is an element of the list of unused atoms it is moved from that list to the list of used atoms.
  3. Otherwise Φ is a non atomic non-γ formula. Formulae of that type may be used only once. Therefore, Φ is removed from the branch.
- `remove_sformula_from_branch/3`
  Remove the given signed formula from the branch. The result is the updated branch without that formula. The formula may be used or new.

The following example shows how the Dewey Numbers are attached to the branches of a tableau.

Example 5.12 *A node of the tableau in figure 5.1 is marked with the Dewey Numbers of the branch which has that node as leaf. The Dewey Numbers are written as a list in Prolog syntax.*

5.5.7 Data Structures for Equality Handling

There are several data structures defined in `datastructures` that are solely used by the modules `complete` and `equality`: `constraint`, `cterm`, `possibility`, `sterm`, `inequality`, `disjunction`, `equality`, `inst` and `add_inf`. In the following we describe only the predicates

{`acc`,`get`}`_`*part*`_of_`*datastructure*`(+Datastructure,-Value)`

that allow to access the value of the various parts of the data structures, or, if a part is an uninstantiated Prolog variable, to instantiate that variable. Although not mentioned, for most of these predicates their counterpart `set_`*part*`_of_`*datastructure*`/3` is defined as well, which allows to assign a new value to a part of a data structure, even if that part has been instantiated before.

[5] It is assumed that the extension is part of a larger conclusion. Each extension of the conclusion will result in a different subbranch. The extensions in the conclusion are numbered from left to right. The number which has to be passed to `add_extension_to_branch/4` is simply the position of the extension in the conclusion.

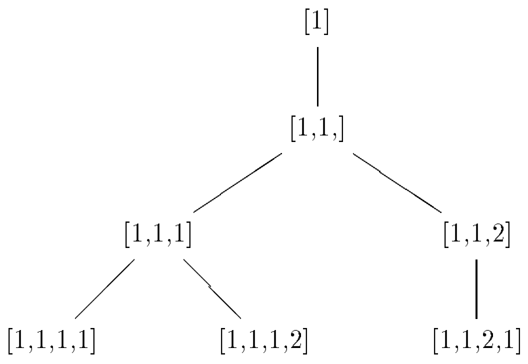

Figure 5.1: Dewey Numbers in a tableau.

5.5.7.1 Representation of Constraints

The data structure `constraint` represents a constraint (Def. 2.34), that can be attached to terms and reduction rules.

The predicates giving access to the parts of the data structure `constraint` are:

- `get_subst_of_constraint`
  The substitution σ that is part of a constraint $c = \langle\sigma, O\rangle$.
- `get_oc_of_constraint`
  The order condition that is part of a constraint $c = \langle\sigma, O\rangle$.

5.5.7.2 Representation of Constrained Terms

The data structure `cterm` represents a constrained term or reduction rule (Def. 2.36).

The predicates giving access to the parts of the data structure `cterm` are:

- `get_prec_of_cterm`
  The precedence according to which lists of `cterm`s are ordered.
- `get_term_of_cterm`
  The term (without constraint).
- `get_constraint_of_cterm`
  The constraint.
- `get_univ_vars_of_cterm`
  The list of variables w.r.t. which the constrained term is universally quantified.
- `get_derived_from_of_cterm`
  The number of the term or rule the `cterm` has been derived from.
- `get_number_of_cterm`
  The number of the `cterm`.

- `get_type_of_cterm`
  The type (`rule` or `term`) of the `cterm`.

- `get_weight_of_cterm`
  The weight of the `cterm` (see Section 5.13.5).

5.5.7.3 Representation of a Possible Rule Application

The data structure `possibility` represents a possibility to apply a constrained reduction rule to a constrained term (which might be a rule too); see Section 5.13.3 and Definition 2.39.

The predicates giving access to the parts of the data structure `possibility` are:

- `get_prec_of_possibility`
  The precedence of the possibility (see Section 5.13.4).

- `get_rule_of_possibility`
  The rule $\mathbf{r}$ that is to be applied.

- `get_cterm_of_possibility`
  The constrained term $\mathbf{t}$ the rule $\mathbf{r}$ is to be applied to.

- `get_position_of_possibility`
  The position l in $\mathbf{t}$ at which $\mathbf{r}$ is to be applied.

- `get_type_of_possibility`
  The type of the possible application (`critical_pair`, `composition`, `simplification`, `term_simplification`, or `non_simplification`).

- `get_unifier_of_possibility`
  The unifier μ that has to be applied.

- `get_new_cterms_of_possibility`
  The list of new constrained terms that are the result of the rule application.

5.5.7.4 Representation of Terms with an Attached Substitution

The data structure `sterm` represents a term with a substitution attached to it.[6] The parts of the data structure `sterm` and the predicates giving access to them are:

- `acc_term_of_sterm/2`
  The term s.

- `acc_inst_of_sterm/2`
  The substitution σ.

- `acc_univ_vars_of_sterm/2`
  The set of variables w.r.t. which s can be seen to be universal.

[6] In this version of $_3\mathcal{T}\!\!^A\!\mathcal{P}$ some of the parts of the data structure `sterm` have become obsolete and are not described here (see the footnote on Page 94).

5.5.7.5 Representation of Unification Problems (Inequalities)

The main parts of the data structure `inequality` are the sets of normal forms for the two sides of an inequality (resp. unification problem) computed so far, and a list of the substitutions already found that allow to solve the problem.

The predicates giving access to the parts of the data structure `inequality` are:

- `acc_left_side_of_inequality`, `acc_right_side_of_inequality`
  The sets of normal forms for the left hand side and the right hand side of the inequality. The sets are implemented as Prolog lists.
- `acc_closings_of_inequality`
  The set of substitutions computed so far that allow to close the inequality (implemented as a list).
- `acc_number_of_inequality`
  The number of the inequality. Actually this is a pair of numbers that includes the number of the disjunction (simultaneous problem) the inequality belongs to.
- `acc_left_counter_of_inequality`, `acc_right_counter_of_inequality`
  The number of terms that have already been added to `left_side` and `right_side`. These counters are needed to generate unambiguous numbers for new terms.

5.5.7.6 Representation of Simultaneous Unification Problems

The data structure `disjunction` represents a disjunction of inequalities (a simultaneous E-unification problem). The parts of `disjunction` and the predicates giving access to them are:

- `acc_expdbl_ineq_of_disjunction`
  The list of inequalities (single unification problems) in the disjunction that are expandable, i.e., that contain a term that an equality can be applied to.
- `acc_inexpdbl_ineq_of_disjunction`
  The list of inequalities in the disjunction that are not expandable.
- `acc_number_of_disjunction`
  The disjunction's number.

5.5.7.7 Representation of Equalities and Demodulators

The data structure `equality` represents equalities and demodulators. If a demodulator is represented, it is oriented from left to right.

The parts of `equality` and the predicates giving access to them are:

- `acc_left_side_of_equality`
  The term on the left hand side of the equality or the demodulator.
- `acc_right_side_of_equality`
  The term on the right hand side of the equality or the demodulator.
- `acc_universal_vars_of_equality`
  The list of variables with respect to which the equality or the demodulator is universal.
- `acc_number_of_equality`
  The number of the equality or the demodulator.

5.5.7.8 Representation of Substitutions

The data structure `inst` represents variable instantiations. It is represented as a Prolog term of the form *variable* `=` *term*, e.g. `X=f(a)`. A substitution is represented by a list of such instantiations.

The predicates `acc_term_of_inst` and `acc_var_of_inst` give access to the parts of an instantiation. In addition, the predicate `make_inst` can be used to build a new instantiation.

5.5.7.9 Representation of Additional Information

The data structure `add_inf` represents information that has to be available in many of the predicates of the module `equality`, and that does not change while a branch is closed. There is only one instantiation of this data structure called `Ai` that is a parameter of most of the predicates. In a way this data structure is an implementation of global variables (relative to `equality`).

The parts of the data structure `add_inf` and the predicates giving access to them are:

- `acc_debug_level_of_add_inf`
  The value of the $_3T^AP$ parameter *eqdebuglevel.*
- `acc_equalities_of_add_inf`
  The list of all equalities on the current branch.
- `acc_demodulators_of_add_inf`
  The list of all demodulators on the current branch.
- `acc_univ_vars_of_add_inf`
  The list of all variables with respect to which one of the atoms on the branch is universal.
- `acc_branch_symbol_of_add_inf`
  An atom identifying the branch. These atoms are of the form `b`n, where n is increased by one whenever a new branch is to be closed.

5.5.8 Data Structures for Achieving Fairness

There are a few predicates gathering information needed for fairness. Here they are:

- `get_label_from_conclusion/2`
  Returns a list of pairs of all the labels at the top level of the formulae with the corresponding signs in the given conclusion. No labels of subformulae of these formulae appear in the list.
- `get_labels_from_extension/3`
  Does the same for an extension (the additional argument is used as accumulator).
- `get_counters_from_labels/2`
  As described in Section 5.7 a counter is attached to every pair of label and sign. This predicate returns the list of counters corresponding to the list of pairs of labels and signs.
- `get_extension_according_to_labels/4`
  Returns the extension which has to be expanded first w.r.t. to the fairness strategy and the remaining conclusion. The decision is based on the list of counters corresponding to the signed formulae in the conclusion.

5.6 Sysdep

The main purpose of the `sysdep` module is to hide system dependent features from the $_3T^AP$ modules. For example, the `append/3` predicate is a built-in in some Prolog implementations, while in others it is a library predicate. Table 5.4 gives an overview of the system dependent predicates handled by the `sysdep` module. These predicates are exported by `sysdep`, their names being prefixed with the string `sysdep_`. For the semantics of these predicates look at the source of the `sysdep` module or your Prolog documentation.

append/3	ask_file/3	change_path_arg/4
contains_term/2	copy_term/2	concat_atom/2
contains_var/2	correspond/4	delete/3
delete_file/1	file_exists/1	flush_output/1
free_of_var/2	genarg/3	gensym/2
is_endfile/1	is_set/1	last/2
list_to_ord_set/2	member/2	merge/4
midstring/4	nth1/3	ord_add_element/3
ord_intersect/2	ord_subset/2	ord_union/3
path_arg/3	pwd/1	remove_dups/2
rev/2	select/3	setof/3
unify/2	union/3	yesno/1

Table 5.4: System dependent predicates.

Other tasks of the `sysdep` module are:

- Importing necessary libraries and exporting the library predicates.
- Definition of the interface predicates to the C foreign language interface and the initialization of the foreign language modules. Cf. your Prolog manual for details.
- Implementation of some low-level predicates, e.g. `reset/0` which resets the prover.
- Implementation of some predicates to access the UNIX environment. Examples are `cp/2` or `ls/0` which simulate their UNIX counterparts.

5.7 Global Variable Management

The modules `globalvars.c`, `globalvars_quintus.c` and `globalvars_sicstus.c` are implemented using the C programming language. `globalvars.c` contains functions to manipulate global counters, flags, switches and other global data structures which would be too expensive to implement in Prolog (via `assert` and `retract`). Its functions are being called through the interface predicates defined in the `sysdep` module. It can be made suitable for different Prolog versions by defining the system dependent constants. `globalvars_quintus.c` includes `globalvars.c` and defines the constants which make the `globalvars.c` module suitable for all of the Quintus Prolog Versions 3.x. `globalvars_sicstus.c` includes `globalvars.c` and defines the constants which make the `globalvars.c` module suitable for the SICStus Prolog Versions 2.1.

In the case of a flag, a switch or a counter there is one global variable defined at the beginning of `globalvars.c` and some functions to read, set or manipulate that variable are implemented. For example, a global variable named `maxcounter` is defined and the functions `set_maxcounter()` and `get_maxcounter()` are supplied to access that variable. For some counters there are additional functions to increment or reset the counter, etc. The implementation of these functions is straightforward, look at the source code of `globalvars.c` for details.

For fairness a counter for each pair of a formula and a sign is needed. `globalvars.c` implements the data structure for these counters. Every subformula may be identified by a unique label, cf. Section 5.5.2.1. These labels are of the form `lxxx` where `xxx` is a positive integer. This integer is used as the first index into an array of dimension 2 which contains the counters. The second index is the sign, the function `signstring2index()` maps the string representation of the sign onto the index range, $[0, \ldots, s-1]$ where s is the number of signs used in the logic. The size of the array is extended automatically if an access to a not existing label happens. The counters may be read, set or incremented. If the array is extended during a read access the counter is initialized to 0, if the extension takes place during an increment operation the counter is set to 1, otherwise it is set to the value supplied by the appropriate argument of the `set_label_counter()` function.

Remark 5.13 *As a matter of fact the index range of the array's second dimension is*

$$0, \ldots, 2(s-1)$$

because there are two counters for each pair of formula and sign. The second counter is used to achieve fairness in the dissolution module.

5.8 Unification

There are two unification predicates implemented in the $_3T^AP$ system. One is for unsorted unification the other for sorted unification. The former predicate, `unify_terms/3`, is implemented in the `sysdep` module the latter, `sorted_unify/2`, in the `unification` module. Of course, both predicates implement a sound unification with occur check.

5.8.1 Unsorted Unification

The `unify_terms/3` predicate in `sysdep` tries to unify copies of its first two arguments. If this is possible the third argument is bound to the unifying substitution. Otherwise, the predicate fails. Unification is done by `sysdep_unify/2` which in the Quintus Prolog version uses the Quintus Prolog library predicate `unify/2` for sound unification. In the SICStus Prolog version a copy of the Quintus Prolog library predicate `unify/2` is used.

5.8.2 Sorted Unification

The predicate `sorted_unify/2` may be used to unify two sorted terms. The additional predicates `sorted_unify_check/3` and `sorted_unify_check/4` may be used to check for sorted unifiability. With the latter predicate it is possible to specify an ambiguity flag which—if set to `yes`—says that the sort hierarchy is a tree or—if set to `no`—says that the sort hierarchy is a directed acyclic graph. If the sort hierarchy is known to be a tree the test for subsorts is easier, Prolog unification of the sorts in the representation described in Section 5.5.1 suffices. Otherwise a more elaborated—but simple—test for sort compatibility has to be done, cf. the source of `unification` for details.

Example 5.14 *Consider the following sort hierarchy:*

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b \arrow[dr] & & c \arrow[dl] \\ & d & \end{tikzcd}$$

Albeit a and b are not subsorts of one another they are compatible because there is a common subsort, namely d.

Except for the subsort test sorted unification is implemented in much the same way as the library predicate, which is a straightforward implementation of Robinson's algorithm with occur check.

5.8.3 Special Treatment of Universal Formulae

Some special care must be taken when terms are to be unified that contain "universal variables" (see Section 2.5. The terms $f(x)$ and $f(g(x))$, that are both universal with respect to x, are unifiable albeit the occur check fails. This is true, because any of the terms may be regenerated with different variables, say y, and $f(y)$ and $f(g(x))$ or $f(x)$ and $f(g(y))$ unify.

5.9 Declarations

The main purpose of the `declarations` module is the declaration of the logic's signature. In this module the representation of signs and connectives is specified. Also, the signs' semantics is defined here by declaring which signs are complementary and which is the axiom and query (theorem) sign. Likewise, some declarative aspects of the `rules` module are handled by `declarations`. These are the definition of the pairs of connectives and sign which are self-contradictory, i.e. where no rules are defined. Finally, some internal declarations, e.g. which symbol should be used for the equality sign and initializations are implemented in this module.

5.9.1 Declaration of the Signature Used in a Logic

5.9.1.1 Connectives and Quantifiers

Some facts and predicates in `declarations` specify the internal and the external representation of the logic's connectives and quantifiers. The symbols used for the external representation of connectives are supplied by the `get_ext_op_list/1` fact. The argument position is filled by a list of these; e.g., `get_ext_op_list([v,&,-,=>,<=>,all,ex,<=])` defines the external symbols for the two-valued version of ${}_3T^AP$, cf. Table 3.2. The list of the corresponding internal representation of the connectives is given by `get_int_op_list/1`. For the above example `declarations` contains `get_int_op_list([dis,con,sneg,imp,equi,all,ex,pmi])`, cf. Table 5.2.

Remark 5.15 *The correspondence between the internal and the external representation of the connectives is given by the position of the symbols in the lists. I.e. the first symbol in the external connectives list is internally represented by the first symbol in the internal connectives list and so on.*
The predicate `get_corresponding_operator/2` *implements this correspondence.*

Remark 5.16 *Please note that the external symbol for the universal and existential quantifier is defined as* `all`, `ex` *resp., although the syntax definition in Chapter 3 says that they are written as* `forall`, `exists` *resp. This is no error!*

The type of the connective is identified by the following predicates.

- `get_ops_with_a_left_fma/1`
  This predicate specifies that the connective has a left subformula. In the above example `declarations` will contain `get_ops_with_a_let_fma([dis,con,sneg,imp,equi,pmi])`.

  No quantifiers are listed here!

- `get_ops_with_two_fmas/1`
  The connectives which have a right subformula are in this list, too. Again, no quantifier is allowed here.

- `get_unary_ops/1`
  Define the unary connectives. The symbol `atfma` is included here, see Section 5.5.2.2. The two-valued `declarations` contain `get_unary_ops([atfma,sneg])`.

- `get_int_quantor_list/1`
  This predicate defines which internal symbols are quantifiers.

- `get_ext_quantor_list/1`
  This predicate defines which external symbols are quantifiers.

5.9.1.2 Signing

The list of signs used by $_3T^AP$ is defined by the `get_int_sign_list/1` predicate. Its argument is the list of signs used by $_3T^AP$. For example, the two-valued version of `declarations` contains the fact `get_int_sign_list( [tSign,fSign] )`.

5.9.2 Declaration of Complementary Signs

There are a couple of facts in `declarations` which define complementary signs. There is one fact of the `is_complementary_sign/2` predicate for each **ordered** pair of signs.

Remark 5.17 *The predicate has to be* **explicitly** *defined as* **symmetric** *by including two clauses for each unordered pair of signs!*

There are just two clauses for the two-valued version of $_3T^AP$:
`is_complementary_sign(tSign,fSign)` and `is_complementary_sign(fSign,tSign)`.

The predicate `is_linking_sign/2` succeeds if and only if any two complementary atoms with these signs imply a link. For the classical two-valued logic and the three-valued logic used in the three-valued version of $_3T^AP$ this predicate is identical to `is_complementary_sign/2`.

The signing of axioms and the theorem is defined by `get_signing/2`. The first argument specifies the axiom sign and the second argument is the query sign, i.e. the sign of the theorem which has to be proved (e.g. `get_signing(tSign,fSign)` for the two-valued $_3T^AP$). If the set of truth values is partitioned into the set of designated and non-designated truth values, i.e. if there aren't any truth values which are neither designated nor non-designated, then the axiom sign is the sign representing the union of all designated signs and the query sign is the union of all non-designated signs which is the complement of the former set.

5.9.3 No Rule defined

The predicate `no_rule_defined/2` succeeds only for pairs of connectives and signs which are self-contradictory. The module `rules` exports a dummy-rule for those cases, see below. For example, the range of the weak negation[7] of the logic implemented in the three-valued version of $_3T^AP$ does not include the truth value corresponding to `uSign`, hence `no_rule_defined( wneg,uSign ) :- !.` is included into the `declarations` module for that logic. As with any rule, see below, a cut is added to the clause to ensure determinism and to eliminate unnecessary choice points.

5.9.4 γ-Formulae

`is_gamma_formula/2` defines which formulae are to be treated as γ-formulae. For example, the two-valued version of `declarations` contains the following clauses:

```
is_gamma_formula(tSign,all).
is_gamma_formula(fSign,ex).
```

In the many-valued case all quantifier rules that generate a free variable in some extension have to be classified as γ-formulae, since they must be applied an indefinite number of times in order to guarantee completeness.

5.9.5 Output-Utility Support

There are two predicates in the `declarations` module for support of the $_3T^AP$ utility which translates a proof protocol into LaTeX syntax.

- `get_LaTeX_op_list/1`
  The only argument position is filled by the list of LaTeX symbols which should be used when a connective is translated. For example

  ```
  get_LaTeX_op_list([vee,wedge,-,supset,
                     leftrightarrow,forall,exists,subset]).
  ```

  specifies that the symbol "supset" ($\supset$) should be used to translate `imp`. The correspondence is by position.

- `get_LaTeX_sign_list/1`
  Specifies the translation of the signs.

5.9.6 Internal Declarations

- `get_sort_op(':')`
  defines the colon to be the sort operator, i.e. the symbol separating a term from its sort, cf. Section 5.5.1 for details.

- `get_equals(=)`
  defines the equality sign to be the symbol for the equality predicate.

- `get_demodulates('==')`
  defines the symbol == to be the name of the demodulator predicate.

[7] The connective is seen here as a mapping from the set of truth values into itself.

5.9.7 Initialization

The predicates `initialize_prover/3` and `initialize_variables/0` may be used to reset $_3T^AP$. The global counters are reset here and the variables and switches are set to their default values.

5.10 Rules

The `rules` module defines the semantics of the logic it carries out proofs in. The most important predicate implemented in this module is `rules/8`. There is a clause for each pair of operator $\circ$ (logical connective or quantifier) and sign σ specifying the rule which may be applied if a formula $p \circ q$ is encountered with sign σ. The remaining parts of `rules` are concerned with the substitution mechanism and support of lemma generation. The auxiliary predicates are not discussed in detail. Their straightforward implementation is documented in the source of the `rules` module.

5.10.1 The Rules Predicate

The arguments of `rules/8` are as follows:

1. The internal representation of the operator (connective or quantifier), e.g. `con`.
2. The sign of the formula. I.e. `tSign` or `fSign` in the classical two-valued case or any other sign declared in `Declarations`.
3. The branching factor of the rule. The branching factor is the number of subbranches which are generated by an application of that rule, i.e. the number of extensions in the conclusion.
4. The total number of formulae generated by an application of the rule.
5. The formula to apply the rule to. The only purpose of this argument is to identify the subformulae. In the example from Item 1 this argument would be `con( Fma1,Fma2,_ )`. Now the extension can refer to the left subformula as `Fma1` and to the right subformula as `Fma2`, see below.
6. The list of variables with respect to which the formula is universal (before rule application).
7. The list of variables with respect to which the formula is universal after rule application. For non-branching rules these lists are generally equal. For branching rules the former list is anonymous in most cases while the latter is empty. γ-rules are exceptions to that.
8. The conclusion of the rule. As usual the conclusion is represented as a list of extensions, cf. Section 5.5.5. The extension is a list of pairs consisting of a sign and a formula. The pair is implemented as a list of two elements. See Examples 5.18, 5.19 for details.

The rules have to be deterministic, i.e. there is only one rule defined for each pair of sign and formula and every clause of `rules/8` is terminated by a cut. To achieve better readability of the rules each extension is in an extra line. It is a good idea to adopt this style when editing the `rules` module.

The following examples are taken from the conjunctive rules of the two-valued version of the $_3T^AP$ system.

Example 5.18 *The rule is of type β, i.e. branching. There are two extensions (2nd argument) with one formula each, which results in a total of two formulae (3rd argument). The rule destroys any "universal variables" (7th argument is* `[]`*).*
```
rule(con,fSign,2,2,con(Fma1,Fma2,_),_,[],
      [[[fSign,Fma1]],
      [[fSign,Fma2]]]) :- !.
```

Example 5.19 *The rule is of type α, i.e. non-branching. There is one extension (2nd argument) with two formulae (3rd argument). The set of "universal variables" for the formula remains unchanged (6th and 7th argument).*
```
rule(con,tSign,1,2,con(Fma1,Fma2,_),Univ_vars,Univ_vars,
      [[[tSign,Fma1],[tSign,Fma2]]]) :- !.
```

5.10.1.1 Dummy Rules

In many-valued logics[8] it is possible that no rule is defined for an operator $\circ$ and a sign σ because $v(a \circ b) = v_\sigma$ or $v(\circ a) = v_\sigma$ does not occur where v_σ is the truth value corresponding to σ and v is a valuation. Such signed formulae are self-contradictory and there is no rule defined for those. Nevertheless, the `rules/8` predicate needs a dummy rule for them.

Example 5.20 *The truth-value corresponding to* `uSign` *is not in the range of the j_u operator* [9] *(partial affirmation,* `ju` *in $_3T^AP$ notation). Therefore no rule is defined for $\{U\}j_u a$. The dummy rule in the* `rules` *module of the three-valued version of $_3T^AP$ looks like this:*
```
rule(ju,uSign,0,0,ju( _,_ ),[ ]) :- !.
```

5.10.1.2 Quantifier Rules

The quantifier rules are of two types: γ- or δ-rules. In the case of an application of a γ-rule to a formula Φ the quantified object variable is replaced by a Prolog variable. A δ-rule application yields a formula where the former quantified object variable is substituted by a Skolem function. In the two-valued version the liberalized δ-rule from (Beckert *et al.*, 1993) is used. Instead of generating a new Skolem function symbol for each δ-rule application, $_3T^AP$ uses function symbols `sko_`n, where n is the label of the δ-formula to be Skolemized. The γ-type formulae may be identified by the `is_gamma_formula/2` predicate in the `declarations` module.

Example 5.21 *The signed formula $\{T\}\forall x \Phi$ is of type γ. The* `rules` *module contains the following clauses:*
```
rule( all,tSign,1,1,all( VarList,Fma,_ ),Univ_vars,New_univ_vars,Conclusion ) :-
     nonvar( Fma ),
     do_gamma( VarList,tSign,Fma,Add_univ_vars,Extension ),
     Conclusion = [Extension],
     sysdep_append(Add_univ_vars,Univ_vars,New_univ_vars),
     !.
rule(all,tSign,1,1,_,_,_,_ ) :- !.
```
The substitution is performed by the `do_gamma/5` *predicate where* `Varlist` *is the list of object variables bound by the universal quantifier. The sign (2nd argument to* `do_gamma/5`*) is used to sign the formula in the resulting extension.*

[8] If 0-ary operators are present this may already happen in two-valued logic.

[9] j_u is seen here as a mapping from the set of truth values into itself.

An example for a δ-rule is similar and may be obtained from the source code of the `rules` module.

The above distinction between γ- and δ-type formulae is not unique. There are formulae which are as well of type γ as of type δ. Quantified `uSign`ed formulae in the three-valued logic used by $_3T^AP$ are examples. There is a variant of the `do_delta/4` and `do_gamma/4` predicate for those, `do_delta_u/6`, which combines the actions taken by the former predicates. `do_delta_u/6` generates a two-element extension.

Remark 5.22 *The signed formulae of this hybrid type must be declared as γ-type formulae by the* `is_gamma_formula/2` *predicate in the* `declarations` *module, cf. Section 5.9.4.*

Remark 5.23 *In the many-valued versions of $_3T^AP$ the mechanism of universal formulae is not used. The* `rules/8` *predicate is only an interface to* `rules/6` *in the many-valued* `Rules` *modules and* `do_gamma` *or* `do_delta` *are of arity four.*

5.10.2 Supporting Lemma Generation

The predicate `get_lemmata/6` is used to decide which formulae may be added to an extension as lemmata. The usage of that predicate is discussed using the following example.
`get_lemmata(con,uSign,con(_,Fma2,_),[1],2,[[tSign,Fma2]]) :- !.`
The meaning of the clause is: If we are expanding the tableau using the rule for `uSign`, `con` and the first extension has been closed already (`[1]` at the 4th argument position) the lemma `[[tSign,Fma2]]` may be added to the second extension (`2` in the 5th argument position). Cases not considered by the `get_lemmata/6` clauses are covered by the dummy lemma predicate
`get_lemmata(_,_,_,_,_,[])` which is always included.

To remove lemmata causing a branching of the tableau the predicates `get_lemmata_alpha/6` and `remove_branching_lemmata/2` are supplied. Cf. the source of `rules` for details.

5.11 Dissolution

The module `dissolve` is an implementation of the dissolution rule for restricted to tableau-based theorem-proving (Murray & Rosenthal, 1990b). The general dissolution rule is described in Section 2.2 and, in greater detail, in (Murray & Rosenthal, 1990a). See also the following section.

The module `dissolve` exports the following two predicates:

1. `treat_dissolution_part`

which is called by `close_branch`. It calls in turn the predicate `dissolve_branch` which controls the whole dissolution module, and afterwards distinguishes different cases depending on the result of dissolution application (and non-dissolution application respectively). The second predicate

2. `check_dissolution_result`

is called by `close_branch` in order to shorten the module `main` and is relatively unimportant.

5.11.1 The Dissolution Rule

In (Murray & Rosenthal, 1990a; Murray & Rosenthal, 1993) Murray & Rosenthal stated a specialized dissolution rule suitable for the method of analytic tableaux. This rule[10] (shown below) is implemented in $_3T^AP$, but only usable in the two-valued version because it is only defined for classical logic.[11]

$$\begin{array}{c} \begin{array}{ccc} \begin{array}{c}A\\ \wedge\\ U\end{array} & \vee & W \end{array} \\ | \\ \begin{array}{ccc}\begin{array}{c}\bar{A}\\ \wedge\\ R\end{array} & \vee & S\end{array}\end{array} \longrightarrow \begin{array}{c} W\\ \wedge\\ \begin{array}{ccc}\begin{array}{c}\bar{A}\\ \wedge\\ R\end{array} & \vee & S\end{array}\end{array} \bigvee \begin{array}{c}A\\ \wedge\\ U\\ \wedge\\ S\end{array}$$

A and $\bar{A}$ are literals, U, W, R and S are subformulae of arbitrary type. During dissolution application the two formulae on the left, which are to be thought on the same branch, are replaced by the dissolvent on the right.

One can easily see that dissolution takes place in the two outermost (syntactic) levels of a formula.

There are several special cases of this dissolution rule:

1. If W (or S, by symmetry) is the empty formula (i.e. does not exist), then the subformula on the right containing W (S) vanishes (this will be called *simple* dissolution rule).
2. If both, W and S are empty formulae, dissolution will close the current branch.
3. If A and $\bar{A}$ are literals within the *same* formula (note that they must occur in the two outermost levels to be detectable), a modification of the above rule is applied. Very often, application of this special rule leads to a closure of the current branch.

5.11.2 Passes

The module `dissolve` is divided into six passes in order to facilitate its understanding and to give it structure.

Pass 0 converts the $_3T^AP$ data structure `sformula` (cf. Section 5.5.3) into list representation. This conversion only takes place in the outer two levels sformula, because dissolution does not need to look at more deeply nested formulae (see Section 5.11.1).

Pass 1 tries to split all sformulae into the form

$$(A \wedge U) \vee W$$

Pass 2 computes *all* possible links (see Section 5.11.3).

Pass 3 additional unification check for preservation of first-order soundness. (see Section 5.11.3)

Pass 4 treats dissolution fairness handling (see Section 5.11.4).

Pass 5 selects the "best" link according to the implemented heuristics (see Section 5.11.6) and applies dissolution rule on it.

[10] We mean that rule when speaking of the dissolution rule from now on.

[11] A generalization to many-valued logics is possible along the lines sketched in (Hähnle, 1992c).

5.11.3 Computation of Links

This part, containing Passes 1 and 2 as described above, is relatively expensive and one of the main reasons for the slowness of $_3T^AP$ with dissolution. It is controlled by the following two predicates:

```
get_all_candidates_for_dissolution
get_all_links
```

The first one contains Pass 1 and is the only part of the whole dissolution module using backtracking, i.e., there exists one predicate `check_lformula` that always fails in order to initiate backtracking. The resulting splitted sformulae are stored in a dynamic list.

The second predicate tries to "combine" every link candidate with each other using a double recursion. It is important to note that in this pass *all possible* literals A and $\bar{A}$ are checked for unifiability but this is only a *weak* unification check due to the internal Prolog dynamic data structure handling. It is theoretically possible that in this pass links $(A\,,\,\bar{A})$ are found whereby A and $\bar{A}$ are weakly but not *strongly* unifiable. Such unsound links are removed in Pass 3 (strong unification check).

5.11.4 Fairness Handling

Some first-order proofs, as for example that of Pelletier's 38th and 46th problem, demand fairness considerations within the dissolution module. Two predicates were implemented to handle fairness:

5.11.4.1 Choice of the "Fairest" Link

If the analysis part in the dissolution module found several candidates for dissolution (i.e. more than one possible link) those link must be chosen which is the *fairest* one, that is, whose literals have least been dissolved upon.

This is a heuristic problem because up to this point every literal of a link $(A\,,\,\bar{A})$ may have been dissolved with several others. To solve this problem every atom is provided with a special label (these are managed in a global hash table implemented in C) which refers to a counter that can be incremented during a tableau proof.

In particular for longer proofs the combination of the following two heuristics proved to yield good results (short runtimes due to fair selection):

1. add to each label counter of A and $\bar{A}$ two and multiply the results. The fairest candidates are those with the lowest product.
2. when dissolving upon a link $(A\,,\,\bar{A})$ increment only the *smaller* label counter of A and $\bar{A}$.

5.11.4.2 Consideration of Dissolved Links

Keeping track of the already dissolved upon links *on the current branch* is very important for preservation of strict first-order completeness. To achieve this a special list `Dissolved_links` is maintained in the predicate `close_branch` (module `main`). The module `dissolve` receives this list as a parameter when called by `close_branch`. If `dissolve` performs at least one dissolution step, say upon a link $(A\,,\,\bar{A})$, two possible modifications can be made on `Dissolved_links`:

1. the link $(A\,,\,\bar{A})$ has not been dissolved upon, yet. Then it is added to `Dissolved_links` and a counter associated with it is set to one.
2. the link $(A\,,\,\bar{A})$ already exists in `Dissolved_links`. Then its counter is increased by one.

Now these old links are used as follows: Before performing a dissolution step all links whose counter is equal to the global variable *dissbound* are removed. That means on the current branch every link can only be dissolved upon a certain number of times. This is important for links stemming from former γ-formulae. Hence, *dissbound* is relatively mighty and its analogy to the global variable *maxcounter* is easy to see (cf. Appendix B).

5.11.5 Facilitation of Understanding

Due to the synthetic nature of the dissolution rule[12] the utility moreTab has difficulties in dealing with it (see also Section 6.4).

Therefore, a support predicate `mark_sformula` is implemented in `dissolve` that shows the atoms A and $\bar{A}$ of the currently dissolved link marked with `**` and `##`, respectively. This can be seen when setting the global variable *dissdebuglevel* to a value of greater than three.

Then both literals of the dissolved link are shown with their marks, as well as the two signed formulae where they come from and the new, synthesized formula. With the help of these marks the dissolution step can more easily be recognized. The marks are only added for output and will not appear in the internal representation of formulae later on in the proof.

When the global switch *disscomplexity* is set to `on` (see Section 5.11.7.1) and a dissolution pair has been cycled the reasons and the result are nice to see in the marked formulae.

5.11.6 Dissolution Heuristics

As described in Section 5.11.1 there are some special cases of the dissolution rule. If the analysis part of `dissolve` found several possibilities to dissolve upon there must be chosen one of them. It is selected according to the following priority among the implemented rules in order to close and reduce the actual branch more quickly:

1. Try closing the current branch by dissolution.
2. Apply dissolution within the same sformula.
3. Look for a "simple" (W or S empty) dissolution.
4. Apply the "full" dissolution rule.

5.11.7 Optimizations

At this point two further optimizations of dissolution shall be presented and discussed. Both are switched, i.e. one can choose between a proof using one (or both) of these optimizations or none of them. Thereby, the influence of either on any proofs can easily be seen.

[12] The formula resulting in its application is in general not a subformula of the problem to be proved.

5.11.7.1 Complexity Check

As can be seen in Section 5.11.1 the dissolution rule is asymmetric, i.e., flipping the input formulae to apply dissolution upon in general leads to a different dissolved formula. The subformula W appears in the new dissolved formula two times while S is included only once.

If *disscomplexity* is activated, `dissolve` will check the selected pair and—if necessary—cycle them in order to get the simplest possible new synthesized formula. Hence, the two subformulae (above called) W and S are tested for complexity wrt "normal" tableau rules, that is, the size of their tableau expansion to the atomic level is measured. In first-order logic there is the possibility to produce γ-formulae, i.e., here the complexity can only be guessed. For a complexity check here it is assumed that every γ-formula is applied at once.

The complexity check is only useful for long formulae, i.e., formulae which are given in one long expression instead of a theorem with several (short) axioms. Especially during proofs of hard propositional formulae such as `pigeon3` or `phi4` (cf. Murray & Rosenthal) this check yields a noticeable speed-up.

5.11.7.2 Rule Priority

A rather awkward side effect of the "full" dissolution rule is the fact that the resulting dissolved formula is of β-type, i.e., splitting this formula by "normal" tableau rules will yield two extensions (notice: dissolution only works in classical two-valued logic). On the other hand, performing a "simple" dissolution step ends in a formula of α-type.

In order to deal with this problem one can change the priorities of the dissolution and the α-rule by setting the switch *disspriority* from `diss` to `alpha`. If it is set to `diss`, $_3T^AP$ will apply the dissolution rule as soon and as often as possible (and legal by fairness handling, see Section 5.11.4). Otherwise, the α-rule is given the higher priority and dissolution will—if applicable—only take place if no α-rule is present on the current branch.

5.12 Index, Makekbx, Preproc

5.12.1 Generating a Compiled Knowledge Base

The modules `index` and `makekbx` contain the predicates for generating and handling knowledge bases. `makekbx` exports the predicate `makekbx`, that builds a compiled knowledge base *file*`.kbx` from the compiler's output *file*`.kb`; it also exports the predicate `readkbx` for reading compiled knowledge bases into the workspace.

The command `compkbx`[13] first calls the compiler (see Section 5.15), that generates the internal representation of the formulae in a knowledge base. Its output is written to the temporary file *file*`.kb`.

`compkbx` then calls the predicate `makekbx`, that generates the compiled knowledge base *file*`.kbx`, i.e., it

1. reads all axioms, theorems and sort declarations from *file*`.kb`;
2. adds the "axiom sign" to the axioms and the "denial sign" to the theorems (the signing strategy is defined in the module `declarations`);
3. sorts the axioms according to a heuristic defined in the module `heuristics`;

[13] And, therefore, as well the command `usekbx`, which is a combination of `compkbx` and `readkbx`.

4. computes the index and the link information;
5. writes the completed knowledge base to *file*.`kbx`.

Finally, `compkbx` deletes the temporary file *file*.`kb`.

To carry out the first three of its tasks, i.e., reading the formulae and sort declarations, adding the signs, and sorting the axioms, `makekbx` calls the predicate `readtcplus`.

The axioms are sorted by the alphabetical order of the prefixes that are computed for each axiom by the predicate `heuristics:get_fma_sort_prefix`.

These prefixes are of the form *branch_number*-*succ_number* (`0-0` for atoms). Based in that, the following heuristic is implemented: Prefer axioms that have

1. a smaller branching factor;
2. a smaller number of successor formulae.

`readtcplus` uses the predicate `internal_set` to write the sorted axioms and theorems and the sort information to the workspace.

After that the predicate `genindices` is called by `makekbx` to compute the index and the link information and to add it to the workspace (cf. Section 5.12.3).

Then the computed knowledge base, consisting of all axioms, theorems, the index and the link information, and the list of formulae that contain the equality predicate, is written to *file*.`kbx` by the predicate `savekbx`. Finally, the knowledge base is deleted from the workspace.

5.12.2 Reading Compiled Knowledge Bases

The predicate `makekbx:readkbx` is the implementation of the command `readkbx`.

First, an older version of the knowledge base to be loaded that might be in the workspace is removed. Then, the file *file*.`kbx` is opened, and the predicate `readkbx_loop` is called to read the knowledge base into the workspace.

`readkbx_loop` reads a line from *file*.`kbx`, and, unless the end of the file is reached, calls itself recursively. Since each line contains only one entry whose type is denoted by its leading function symbol, the entries can easily be recognized and then be written to the workspace using the predicate `sysdep:internal_set`.

When the knowledge base has been read, the predicate `assert_sort_ambiguity` is called to check the sort declarations for ambiguity.[14]

5.12.3 Computing the Link Information

5.12.3.1 Theoretical Aspects of Using Links

Obviously, it would be of great advantage, if it were possible, to realize that certain formulae on a branch are of no use for closing the branch, and therefore can be deleted. Fortunately, at least some such formulae can be recognized, namely those that are neither linked to another formula on the branch, nor to a formula in the knowledge base, that might be put on the branch later on, nor contain the equality predicate. They can never participate in the closure of a branch.

[14] If the sort declarations are not ambiguous, i.e., if from each sort there is only one path to the sort `top`, a simpler—and faster—unification algorithm can be used.

Definition 5.24 *A signed formula $S'\phi'$ is called an* immediate descendant *of a signed formula $S\phi$ if it can be derived from $S\phi$ by a single application of a tableau rule—including lemma generation.*

$S'\phi'$ is called a descendant *of $S\phi$ if there are signed formulae*

$$S'\phi' = S_0\phi_0, S_1\phi_1, \ldots, S_n\phi_n = S\phi, \quad (n \geq 0)$$

such that $S_i\phi_i$ is an immediate descendant of $S_{i+1}\phi_{i+1}$ $(0 \leq i \leq n-1)$.

Remark 5.25 *The "is descendant of" relation is the reflexive and transitive closure of the "is immediate descendant of" relation.*

Definition 5.26 *Two signed formulae $S_1\phi_1$ and $S_2\phi_2$ are* linked *if there are atomic formulae $S_1'p(\mathbf{t})$ and $S_2'p(\mathbf{s})$ that are descendants of $S_1\phi_1$ and $S_2\phi_2$ respectively and that have the same predicate symbols and complementary signs (i.e., $S_1' \cap S_2' = \emptyset$).*

To check, whether a knowledge base has a link, all atoms have to be computed that can be generated by the application of tableau rules to it. The information about links can be pre-compiled, thus, it has only to be done once, whereas, if information about links is not present, tableau rules may be applied to a useless formula on every branch on which it occurs, and this is much more expensive.

5.12.3.2 The Index and the Link Information

There are two different ways of taking advantage of the knowledge about links between formulae:

1. impose a restriction on the formulae that are put onto a branch;
2. remove unlinked formulae from a branch.

$_3T^AP$ always employs Method 1. To use it, only the relation "is descendant of" between formulae in the knowledge base, i.e., axioms and theorems, and their atomic descendants must be known. This "partial" link information, called index, is included in every compiled knowledge base.

Since, for large knowledge bases, it may take very long to compute the complete link information that is necessary to employ Method 2, it is only pre-compiled if *removeunlinked* is switched on. Then, a list of all existing links between all the descendants of all formulae in the knowledge base is computed and included in the compiled knowledge base. Since formulae containing an equality or the demodulator predicate must not be deleted if *equality* is switched on, a list of all such formulae is computed as well.

5.12.3.3 Implementation

The predicate `genindices` computes the index and, if *removeunlinked* is on, the complete link information.

To do this, first, the predicate `datastructures:get_all_indices` is called, that returns a list of the names of all axioms and theorems in the knowledge base. This list is handed to `genidx`.

`genidx` recursively computes all descendants of all axioms and theorems by applying the tableau rules defined in the module `rules`. While doing this it computes

1. a complete list of all occurring “is immediate descendant of” relations;
2. all index entries, i.e., the “is descendant of” relation between axioms and theorems and their atomic descendants.

`genidx` cannot generate the complete link information, because at each level of the recursion only the immediate parent formula and the name of the initial signed formula (i.e., the axiom or theorem that was the starting point of the recursion) are known. The former is used to generate a “is immediate descendant of” entry, the latter to generate an index entry whenever the atomic level is reached.

Since the tableau rules as well as the lemma generation rules are analytical, i.e., only yield new formulae that are subformulae of formulae already on the branch[15], and since subformulae are marked with a label, all occurring “is immediate descendant of” relations can be represented by two label/sign pairs.

`genidx` keeps a list of all label/sign pairs (and hence, implicitly of all subformulae) that a tableau rule has already been applied to. This list is updated and passed on when `genidx` calls itself recursively. It is used to avoid processing a label/sign pair more than once that occurs multiply in a conclusion of a tableau rule. Therefore, the complexity of the computation depends on the size of the tree representing the structure of the processed formula and not on the size of the tree that is generated by applying to each subformula once the corresponding tableau rule.[16]

Nevertheless, index entries can be generated multiply, since a formula might contain identical subformulae in different places (with different labels). Therefore, the lists of index and “is immediate descendant of” entries are treated as sets, i.e., duplicates are not included.

The complete lists are written to the workspace with predicates `assert_idx`, `assert_subform`.

Then, the predicate `assert_links_and_cont_eq` is called to compute the complete link information and a list of all occurring formulae that contain the equality or the demodulator predicate sign.[17]

The implementation of `assert_links_and_cont_eq` is based on the enumeration predicates `assert_all_links` and `assert_all_cont_eq`, that always fail. They use Prolog's backtracking to assert a list of *all* links and *all* formulae containing the equality predicate to the workspace.

Starting from the links between atomic formulae (that can be found using the index entries written to the workspace by `genidx`) and the formulae at hand from which a given formula can be derived (these have been written to the workspace by `genidx`, too), all links can be efficiently computed.

In a similar way, all formulae containing the equality or the demodulator predicate symbol can be computed starting from the atomic formulae that are equalities or demodulators.

Remark 5.27 *The dissolution rule is not analytical. It can generate completely new formulae. Information about links of such new formulae is not computed and not included into a compiled knowledge base. These new formulae are, therefore, never deleted from a branch, even if* removeunlinked *is switched on.*

[15] Of course, there is no restriction on the truth value signs of new formulae.

[16] In classical two-valued logic, both are of the same size; but in certain multiple-valued logics, the latter can grow exponentially in the length of the formula.

[17] This additional information is only computed when *removeunlinked* is switched on.

5.12.4 Syntax of Compiled Knowledge Bases

Each line of a compiled knowledge base *file*`.kbx` contains one entry consisting of a single Prolog term, whose leading function symbol denotes its type:

`sf(`*name*`,`*sign*`,`*formula*`)` A "signed formula", i.e., an axiom or a theorem. *name* is the name, *sign* the truth value sign, and *formula* the formula itself (in internal representation).

`th(`*theorems*`)` A list of the names of those "signed formulae" that are theorems.

`i(`*atom*`,`*sign*`,`*name*`)` An index entry with the meaning: the atom *atom* with sign *sign* is a descendant of the "signed formula" *name*.

`s(`*sortpath*`)` *sortpath* is a list of sorts representing a sort path.

`link(`*label_1*`,`*sign_1*`,`*label_2*`,`*sign_2*`)` A link entry with the meaning: the formula labelled *label_1* with sign *sign_1* is linked to the formula labelled *label_2* with sign *sign_2*.

`cont_eq(`*label*`)` This entry means that the formula labelled *label* contains the equality or the demodulator predicate symbol.

In addition, a compiled knowledge base contains comment lines starting with `%`.

The temporary files *file*`.kb` generated by the compiler have a similar syntax. However, they contain no link information and the formulae are not signed. There are three different entry types:

`a(`*name*`,`*formula*`)` The axiom named *name*; *formula* is its internal representation.

`t(`*name*`,`*formula*`)` The theorem named *name*; *formula* is its internal representation.

`s(`*sortpath*`)` *sortpath* is a list of sorts representing a sort path.

5.12.5 Pre-processing Formulae

The module `preproc` exports predicates for pre-processing formulae; it uses a method based on removing "anti-links" (Beckert *et al.*, 1994). In Version 3.0 of $_3T^AP$ this is an "undocumented feature"; it is only prototypically implemented, and only experienced users should set the switches *flattenformulas* and *removeantilinks* to on, that control pre-processing.

If, despite this warning, you want to use the module `preproc`, please consult the comments in the source code for a description.

5.13 Complete, Equality

5.13.1 Overall Structure of Equality Handling

The module `complete` is an implementation of the method for handling equality in tableaux described in Section 2.6, based on the completion-based algorithm from Section 2.6.6 for solving mixed E-Unification problems.[18] It exports the main predicate

```
    close_branch_with_completion(+Branch)
```

[18] The module `complete` can be used stand-alone for solving mixed E-unification problems; see Section C.7.

that is called by `close_branch` if a branch is exhausted and could not be closed yet (provided *equality* is switched on).

The module `equality` provides additional predicates which are called by the module `complete`, in particular those for extracting equalities and unification problems from a branch.

If `complete` succeeds to close the branch using equality, the necessary free variable substitutions are applied. When backtracking occurs, further substitutions are searched for that allow to close the branch.

After the predicate `close_branch_with_completion` has been called with a branch B, first the set $E(B)$ of equalities and the set $\mathcal{P}(B)$ of unification problems are extracted from B (see Section 5.13.2).

After that, the predicate `try_to_close_at_once_else_eq_appl_for_closing` is called. This predicate contains a choice point. It uses all closing substitutions to close the branch B that can be found without equality applications. Such substitutions σ exist if there are inequalities $\mathsf{F}\,(t \approx s)$ on the branch such that σ is an MGU of t and s.

If no (or, after backtracking, no further) such immediately closing substitutions exist, the main loop of `complete` is started; it computes the completion of $E(B)$ and normalizes the terms in $\mathcal{P}(B)$ (see Section 5.13.3). The completion process and the normalization process are combined (see Section 2.6.6.4). The two processes can be separated by switching on *complete_first*; in that case, first a complete set of reduction rules is computed, and after that the complete set is applied to compute normal forms.

If none of the computed unifiers in the ground-completeset $\mathrm{Sat}(\mathcal{C}(\langle E, s, t\rangle))$ (Theorem 2.55), can be used to close the tableau, the orientation of rules in the completion $\mathcal{R}^\infty$ for E is changed, and the inversion is applied to the unifiers computed to far. However, additional solutions are only computed, provided the switch *compute_additional_solutions* is on (default: off).

The search for solutions is limited by several parameters; see Section 5.13.6.

5.13.2 Extracting Equalities and Unification Problems

The sets $E(B)$ (Def. 2.25) and $\mathcal{P}(B)$ (Def. 2.26) are extracted from the branch B by the predicate

```
extract_disjunctions_and_equalities(+Branch,-Disjunctions,
                                    -Closing_inst,-Ai)
```

in module `equality`.

First, the predicate

```
extract_pos_neg_eq_univ_vars_from_branch
```

is called, that extracts a lists of all positive atoms (that are no equalities or demodulators), negative atoms (that are no inequalities), equalities, demodulators and inequalities from the branch, and, in addition, a list of all variables with respect to which one of these atoms is universal. All sort information is removed at this point.

Then, the predicates `build_equalities`, which computes $E(B)$, and `build_disjunctions` are called. For historical reasons, first the data structure "disjunctions of inequalities" consisting of `stermss` is used to represent the unification problems.[19] The predicate `problems_to_cterm_prob` is used to transform them into `cterms`.

[19] In the earlier versions of $_3T^AP$ a method based on computing equivalence classes was used to solve E-unification problems. This method represented terms by `sterm`. Therefore, `sterm` is still used by the part of module `equality`, that extracts equalities and unification problems, and that has not been changed.

For each equality $\mathsf{T}\,(t \approx s)$ on a branch B that is universal with respect to variables $x_1, \dots, x_n$, both $(\forall x_1)\dots(\forall x_n)(t \approx s)$ and $(\forall x_1)\dots(\forall x_n)(s \approx t)$ are added to $E(B)$.

`extract_disjunctions_and_equalities` returns, in addition to $E(B)$ and $\mathcal{P}(B)$, a list of those substitutions that allow to close the branch without any equality applications (for example $\{x \leftarrow a\}$ if the branch contains the inequality $\mathsf{F}\,(f(x) \approx f(a))$).

5.13.3 Complete's Main Loop

The predicate

```
cycle(+System,+Possibilities,+Problems,+Solutions_so_far)
```

implements `complete`'s main loop. It executes one completion or normalization step and then calls itself recursively.

For efficiency it is very important which (fair) completion and normalization procedure is used to solve E-unification problems. $_3T^AP$ uses a heuristic that compares all possible rule applications (both completion and normalization steps). The possible applications of the critical pair rule, the deduction rule, the composition rule, and the simplifications rules (Sections 2.6.6.3 and 2.6.6.4) kept in a list `Possibilities` sorted according to their precedence (see the next section). There is only one list for all completion and normalization sequences computed to solve the different E-unification problems derived from a tableau branch.

`System` is the set of reduction rules computed so far; `Problems` contains the normal forms of the terms in the unification problems that have been computed; `Solutions_so_far` is a list of the closing substitutions already found.

If no rule application is possible, or if the best possibility exceeds a limit (see Section 5.13.6), `generate_further_solutions` is called to apply the inversion of the system of reduction rules to `Solutions_so_far`.

Else, the best possibility is chosen, and according to its type the new rules or terms are computed. After that,

1. the constraints of the new terms or rules are transformed into normal form, such that they only contain simple order conditions (see Section 5.13.10);
2. if possible, i.e., if the rule application is a simplification, the old rule or term is removed;
3. subsumed terms and rules and those with an inconsistent constraint are removed;
4. the new possibilities to apply a completion or normalization rule are computed, their precedence is determined, and they are added to the list of possible applications;
5. possibilities that ceased to exist, because one of the involved terms or rules has been removed, are deleted from the list.

5.13.4 Precedence of Possible Rule Applications

The precedence of possible rule applications is computed by the predicate

```
compute_precedence_of_possibility
```

Suppose M_i $(i = 1, 2)$ is the possibility to apply the rule $\mathbf{r}_i$ to the term $\mathbf{t}_i$ (which might be a rule as well), and, thus, to derive the new terms $\mathbf{t}_i^1, \ldots, \mathbf{t}_i^k$ (that are already in normal form). $G(\mathbf{t})$ is the weight of the term $\mathbf{t}$ (see Section 5.13.5); Index($\mathbf{t}$) is the index of a term or rule, i.e., if Index($\mathbf{t}$) $= n$, then $\mathbf{t}$ is the nth term added to the completion or normalization sequence. Using this notation, the implemented heuristic can be formulated in the following way: the possibility M_1 ist better than M_2 if[20]

1. $\max(\text{Index}(\mathbf{r}_1), \text{Index}(\mathbf{t}_1)) - \max(\text{Index}(\mathbf{r}_2), \text{Index}(\mathbf{t}_2)) \; > \; d$.
2. M_1 is a simplification or composition, whereas M_2 is an application of the critical pair or the deduction rule.
3. $\max(G(\mathbf{t}_1^1), \ldots, G(\mathbf{t}_1^k)) \; < \; \max(G(\mathbf{t}_2^1), \ldots, G(\mathbf{t}_2^k))$.
4. $\max(G(\mathbf{r}_1), G(\mathbf{t}_1)) \; < \; \max(G(\mathbf{r}_2), G(\mathbf{t}_2))$.
5. $\min(G(\mathbf{r}_1), G(\mathbf{t}_1)) \; < \; \min(G(\mathbf{r}_2), G(\mathbf{t}_2))$.
6. $\max(\text{Index}(\mathbf{r}_1), \text{Index}(\mathbf{t}_1)) \; < \; \max(\text{Index}(\mathbf{r}_2), \text{Index}(\mathbf{t}_2))$.
7. $\min(\text{Index}(\mathbf{r}_1), \text{Index}(\mathbf{t}_1)) \; < \; \min(\text{Index}(\mathbf{r}_2), \text{Index}(\mathbf{t}_2))$.

The first criterion assures fairness of the procedure (Def. 2.47 and 2.50) for all computed completion and normalization sequences. The value of d is chosen high enough, such that the criterion applies only very rarely in practice (by default: $d = 300$).

The second criterion keeps the number of rules and terms small, because after simplifications and compositions, the old term (resp. rule) is removed.

It is essential to take the term weight into concern in some way or the other. However, many experiments were necessary to develop Criteria 3 to 5.

5.13.5 Term Weight

By default, the weight of a constrainted term $\mathbf{t}$ is the number of function symbols, constant symbols, variables, and logical operators occurring in $\mathbf{t}$ (including its constraint).

If the switch *weight_left_only* is on (default: off), the weight of a constrained rule does not include its right side, i.e., only the symbols in the left side and in the constraint of the rule are counted.

A different term weight can be defined by changing the predicate `weight` in module `complete`.

5.13.6 Parameters Limiting Completion and Normalization

There are four parameters that limit the completion and normalization process and thus the search for solutions of E-unification problems:

max_solutions_per_branch
: The maximal number of closing substitutions that are computed for a branch using equality (default: 10).

max_rule_cr_number
: The maximal number of applications of the critical pair rule per branch (default: 10000).

[20] The criteria are listed according to their importance. Only if a criterion does not distinguish two possibilities, the next one is taken into concern.

max_rule_simp_number
: The maximal number of applications of the composition and the simplification rule per branch (default: 10000).

max_term_number
: The maximal number of new constrained terms that are derived during the computation for closing a single branch (default: 10000).

These parameters have similar effects as the parameters *maxcounter* and *maxbranchlength*, that limit the expansion of tableaux. On the one hand, their value has to be sufficiently high to prove theorems; on the other other hand, lower values reduce the size of the search space.

5.13.7 The Lexicographic Path Ordering Used

Module `complete` uses a lexicographic path ordering on terms, that is induced by the ordering $>_F$ on function and constant symbols (resp. its transitive closure). By default, $>_F$ is defined by: $g >_F f$ iff

1. the arity[21] of g is greater than that of f, or
2. g and f have the same arity, and g is behind f in the alphabetical order.

This ordering turned out to be suitable for most problems. But, since there are exceptions, the user can change the ordering on function symbols by adding Prolog facts such as

```
precedence(f,g).
```

to the module `complete`. Symbols that are not comparable in the transitive closure of the order defined by `precedence` remain ordered according to the default ordering defined above.

5.13.8 Computing Additional Solutions

If no further solutions to the E-unification problems extracted from a tableau branch can be found, or if one of the limits *max_rule_cr_number*, *max_rule_simp_number*, or *max_term_number* (see Section 5.13.6) is reached, the predicate

```
generate_further_solutions(+Reduction_system,+Solutions)
```

is invoked, to compute additional solutions.

This is done by changing the orientation of the rules in `Reduction_system` and applying them to the unifiers in `Solutions`. Each new solution is used to close the branch. If no (additional) solution can be found or *max_solutions_per_branch* is reached, `generate_further_solutions` fails.

In theory, computing additional solutions is necessary for completeness of the method (see Section 2.6.6.6). Fortunately, in practice `generate_further_solutions` has to be called only very rarely to prove a theorem. Therefore, this is only done if *compute_additional_solutions* is on.

[21] Here, constants are treated as functions of arity 0.

5.13.9 Handling Substitutions

5.13.9.1 Canonical Representation

Two unifiable terms s and s' always have an MGU σ that is canonical, i.e., that meets the following condition:

Definition 5.28 (Canonical Substitution) *A substitution*

$$\sigma = \{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\},$$

is called canonical, if the variables x_i do not occur in any of the terms t_j $(1 \leq i, j \leq n)$.

Since all substitutions that occur during equality applications are either the MGU of two terms or are a specialization of an MGU, it is possible to impose the general restriction on substitutions that they have to be canonical. As a result, simpler and faster algorithms can be used for handling substitutions.

The predicate `test_unify_and_give_subst` is used to compute a canonical MGU. It is based on the predicate `unify_terms` exported by the module `unification`. Even if this predicate generates an MGU in an incorrect form, such as `[X=Y,f(a)=Y]`, `test_unify_and_give_subst` computes the correct canonical equivalent `[X=f(a),Y=f(a)]`. To do this it uses the predicates `condense_substitution` and `orientate_substitution`.

5.13.9.2 Combining Substitutions

The predicate `combine_instantiations` computes for two given substitutions σ and σ' their combination, i.e., a substitution τ such that

1. σ and σ' are both more general than τ,
2. τ is more general than all substitutions having Property 1.

If no such substitution τ exists, `combine_instantiations` fails. A predicate `combine_inst_2` is used, that implements the recursive algorithm shown in Figure 5.2 using an accumulator α to iteratively compute the substitution τ. The substitutions σ and σ' have to be canonical (Definition 5.28).

Remark 5.29 *The notation $\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}\nu$, i.e., the application of one substitution to another, is defined as $\{x_1 \leftarrow t_1\nu, \dots, x_n \leftarrow t_n\nu\}$.*

5.13.10 Checking Consistency of Constraints

To implement the completion based method from Section 2.6.6.3, the algorithm for checking consistency of constraints could be used, that has been described in (Comon, 1990). This algorithm, however, is quite complex, which is no surprise because the problem is NP-hard. The reason is, that inconsistencies have to be taken into concern that, for example, stem from the fact that the order condition $(b \succ x) \wedge (x \succ a)$ is only satisfiable, provided there is a constant symbol between a and b (w.r.t. to LPO used).

Since consistency of constraints has to be tested very often, in particular for all subsumption checks, Comon's algorithm is too inefficient for implementation.

combination(σ, σ'):

```
$\alpha := \emptyset$
WHILE $\sigma' \neq id$
    IF $\sigma = \emptyset$
    THEN Choose $x$ such that $\sigma' = \{x \leftarrow t\} \cup \sigma'_{Rest}$
         $\sigma' := \sigma' \setminus \{x \leftarrow t\}$
         $\alpha := \{x \leftarrow t\} \cup \alpha\{x \leftarrow t\}$
    ELSE Choose $x$ such that $\sigma = \{x \leftarrow t\} \cup \sigma_{Rest}$
         IF $\sigma' = \{x \leftarrow t'\} \cup \sigma'_{Rest}$
         THEN IF $t$ and $t'$ are unifiable with MGU $\nu$ and
                 $combination(\nu, (\sigma' \setminus \{x \leftarrow t'\}))$ exists
              THEN $\sigma' := combination(\nu, \sigma' \setminus \{x \leftarrow t'\})$
                   $\sigma := \sigma \setminus \{x \leftarrow t\}$
                   $\alpha := \{x \leftarrow t\nu\} \cup \alpha$
              ELSE FAIL: Combination does not exist
         ELSE $\sigma' := \sigma'\{x \leftarrow t\}$
              $\sigma := \sigma \setminus \{x \leftarrow t\}$
              $\alpha := \alpha \cup \{x \leftarrow t\}$
Output: $\sigma \cup \alpha$
```

Figure 5.2: The algorithm for computing the combination of two substitutions.

The following pragmatic approach is more suitable. It is based on the assumption, that adding new constant symbols to the signature does not do any harm. The advantage of an expansion of the signature is, that a new symbol can—for example—be defined to be between a and b. Instead of really adding new symbols, one can instead ignore inconsistencies such as $(b \succ x) \wedge (x \succ a)$, and check order conditions only for "weak" consistency.

It is comparatively easy to decide, whether an order condition O or, based upon that, a constraint $\langle\sigma, O\rangle$ is weakly consistent, i.e., consistent if the signature is expanded appropriately.

Example 5.30 *Supposed, a and b are the only constant symbols, and $b \succ_{\mathrm{LPO}} a$; f is a minimal function symbol w.r.t. $\succ_{\mathrm{LPO}}$. Then the order conditions*

$$(b \succ x) \wedge (x \succ a), \quad (f(a) \succ x) \wedge (x \succ b), \quad (a \succ x)$$

are weakly consistent, but not consistent. The conditions

$$x \succ x, \quad (x \succ y) \wedge (y \succ z) \wedge (z \succ x), \quad x \succ f(x)$$

are neither weakly consistent nor consistent.

Definition 5.31 (Simple Order Condition) *An order condition O is simple, if it is identical to true or false, or of the form*

$$(s_1 \succ t_1) \wedge \ldots \wedge (s_n \succ t_n) \qquad (n \geq 1) \ ,$$

and for all $(s \succ t) \in O$ (where O is treated as an implicitly conjunctive set):

1. *$s \succ t$ is neither true nor false.*
2. *s or t is a variable.*

3. *There is no $(s \succ u) \in O$, such t is a subterm of u.*
4. *There is no $(u \succ t) \in O$, such that u is a subterm of s.*
5. *If $(u \succ v[s]) \in O$, then $(u \succ v[t]) \in O$.*

A simple order condition is—provided it is not identical to *true* or *false*—consistent on the atomic level (Condition 1); at least one side is a variable (Condition 2); it does not contain any inconsistencies induced by the monotonicity of $\succ$ w.r.t. to the term structure (Conditions 3 and 4); and it is (in a certain sense) transitively closed (Condition 5).

To check whether an order condition is weakly consistent, it is sufficient, to transform it into an equivalent simple order condition (or a set of disjunctively connected simple order conditions, see Lemma 5.32). The reason is that a simple order condition is weakly consistent if and only if it is not identical to *false*.

In addition, simple order conditions are easier to handle; in particular it is easy to compute the negation and conjunction of simple order conditions.

Lemma 5.32 *For each constraint c there is a set*

$$C = \{\langle\sigma_1, O_1\rangle, \dots, \langle\sigma_m, O_m\rangle\} \qquad (m \geq 0)$$

of constraints, such that

$$\mathrm{Sat}(C) = \mathrm{Sat}(c) \enspace ,$$

and the order conditions O_i $(1 \leq i \leq m)$ are simple.

The following algorithm can be used to compute a set C of simple order conditions equivalent to a given order condition $c = \langle\sigma, O\rangle$:

1. First, the order condition O is transformed into disjunctive normal form (DNF) by logical transformations, i.e., into the form

   $$(O_{1,1} \wedge \dots \wedge O_{1,l_1}) \vee \dots \vee (O_{k,1} \wedge \dots \wedge O_{k,l_k}) \qquad (k, l_1, \dots, l_k \geq 1) \enspace ,$$

   where the O_{ij} are atomic order conditions.

2. Then, the set

   $$\begin{aligned} C' &= \{c'_1, \dots, c'_k\} \\ &= \{\langle\sigma, (O_{1,1} \wedge \dots \wedge O_{1,l_1})\rangle, \dots, \langle\sigma, (O_{k,1} \wedge \dots \wedge O_{k,l_k})\rangle \end{aligned}$$

   that is equivalent to c is generated.

3. The constraints c'_i $(1 \leq i \leq k)$ that only contain the logical operator $\wedge$, can be transformed into simple order conditions by applying Condition 5 in the definition of simple order conditions (Def. 5.31) and Conditions 1 to 3 in the definition of lexicographic path orderings (Def. 2.29) as transformation rules. The transformation terminates, if either a simple order condition is generated, or an obvious inconsistency is found, i.e., an inconsistent atomic condition or a violation of Conditions 1 to 4 in the definition of simple order conditions.

All occurring constraints are immediately transformed in this way. Therefore, at no time there are order conditions that are more complex than those that are negations, conjunctions, or instances of simple order conditions.

5.13.11 Applying Demodulators to a Term

The predicate `demodulate_term` applies the demodulators present on the current branch to demodulate a term. If the demodulators form an equality theory that is not noetherian, the algorithm may fail to terminate.

The demodulators are applied in the order they occur on the branch. A demodulator is preferably applied to the whole term; only if that is not possible, it is applied to the term's subterms. If the term has been changed by a demodulator application, the process is started all over and the first demodulator is tried again. This goes on until no further demodulator applications are possible.

A demodulator is not applied if that requires the instantiation of a free variable with respect to which the demodulator is not universal.

5.14 Output, Msg_tap

The module `output` exports the predicates `x_write` and `x_nl`, that are used by all other modules for writing information to the current output stream, predicates for writing most of $_3T^AP$'s data structures, and for displaying error messages. The data structures are stripped of all irrelevant information and typeset in such a way that they can be easily read and understood.

In `msg_tap` $_3T^AP$'s error messages are defined. It is, actually, not a module, but a file included by means of the `consult` predicate.

Mainly, `output`'s predicates are used by the other modules to display debug information.[22] For debugging the modules `completion` and `equality`, predicates are defined that—depending on the value of *eqdebuglevel*—are called at certain points, provided *eqdebuglevel* is set to a value greater than 0.

In addition, `output` contains the predicates for generating a protocol file that can be read by the programs moreTab and tabTEX. These predicates do—in contrary to all other predicates in `output`—not use the predicate `x_write`, but write directly to the protocol file denoted by the parameter *tableauoutfile*. They are called at certain points of a proof provided *tableau_output* is switched on, e.g. the predicate `write_branch_closed` is called whenever a branch has been closed and writes the path of the closed branch to the protocol file. The structure of the protocol information and its syntax is described in Sections 6.3 and 6.6.

Some of the predicates in `output`, such as the predicate `proof` that displays the message "proof found", are called by other modules even if *debuglevel* is set to 0.

`x_readchar(-Char)` is the only input predicate in `output`. It prompts the user for a single character; the character is read from the current input stream.

5.14.1 The Predicates X_write and X_nl

The predicate `x_write/2` is used for displaying all material that is supposed to be written to the standard output[23] (with the exception of error messages, cf. Section 5.14.3). This feature makes it possible to easily redirect $_3T^AP$'s output in whole or in part.

`x_write`'s first argument is the text to be displayed. It is either a single atom or a list of atoms. These atoms are displayed separated by blanks and followed by a newline. The second argument is one of the atoms listed in Table 5.5; it denotes the type of information that is to be displayed.

[22] Debug information is displayed if the parameter*debuglevel* (resp. *dissdebuglevel* and *eqdebuglevel*) is set to a value greater than 0, cf. Appendix B.

[23] I.e., all output that is not written to `proveall`'s statistic file or the protocol file used by moreTab and tabTEX.

Type	Displayed Material of that Type
`result`	The message that a proof has been or couldn't be found, and the statistical information provided with that message.
`lookup`	The information that is being displayed by the predicates `lookup`, `writekb`, `writekbx`, `writeidx` and `writesort`.
`debug`	All debug information displayed if one of the parameters *debuglevel*, *eqdebuglevel* or *dissdebuglevel* is set to a value different from 0.
`help`	The help pages displayed by the `info` command.
`info`	Miscellaneous messages (e.g. "Generating index for *KB*").

Table 5.5: The different types of information displayed by `x_write`.

If $_3T^AP$ is running under IBM's LILOG-KR user interface[24], `x_write` uses the predicate `w_writel` exported by LILOG-KR's module `iw_inout` to write the text to the appropriate LILOG-KR window. Otherwise, if $_3T^AP$ is running standalone, `x_write` uses the predicate `subst_w_writel` to write the text to the current output stream.

If *protmode* is switched on, `x_write` writes all output to the file denoted by *outputfile*.

`x_nl(Where)` displays an empty line. It is an abbreviation for `x_write('', Where)`.

5.14.2 Auxiliary Output Predicates

The following list contains the predicates in the module `output` that are merely auxiliary to other output predicates, or that are not for the output of certain data structures.

`write_list` Writes an arbitrary Prolog list; a newline is inserted after each item.

`write_clist` Writes an arbitrary Prolog list.

`write_remark` Writes text marked as a comment by a leading `%`.

5.14.3 Error Messages

`output` exports the predicate `error_message/1,2`. The first argument (and single argument of `error_message/1`) is an atom denoting the message to be displayed. The second argument is either a single or a list of up to three parameters, that are to be substituted for variables occurring in the definition of the error message.

The messages are defined in `tap_msg` in a LILOG-KR specific format (see the comments in `tap_msg` for a description of that format). Since the messages are not spread all over $_3T^AP$'s modules, they can easily be changed and new messages can be added.

If $_3T^AP$ is running under LILOG-KR, `error_message` is implemented by `display_message/1,2` which is exported by one of the LILOG-KR modules, namely `i_displ`; else `error_message` uses the predicate `subst_display_message/1,2` defined in module `output` for formatting the message and displaying it.

[24] For testing, whether $_3T^AP$ is running under LILOG-KR or not, a fact `is_lilog_version(yes)` (respectively, `is_lilog_version(no)`) is asserted in `boot`.

5.14.4 Predicates for Displaying Data Structures

Predicates exist for most of the data structures described in Section 5.5 to write them in a simplified and more readable form to the output stream. For examples see the following section and Section 7.3.

`write_branch` Writes the used and unused formulae and atoms and the branch's path to the output stream.

`write_extension` Writes a list of the formulae in an extension to the output stream.

`write_sformula` Writes an `sformula` in the form

(*counter*) . *simplified_formula* << *index*

to the output stream. *simplified_formula* is the result of applying the the predicate `simplify_sorted_sformula`, that substitutes sort paths by the corresponding specific sorts. *counter* is the number of γ-rule applications to the formula.

`write_conclusion` Writes a list of all the extensions in a conclusion to the output stream.

`write_new_sterm_list` Writes a list of `sterms`.

`write_equality_list` Writes a list of equalities or demodulators to the output stream. The n^{th} equality $(\forall u_1)\ldots(\forall u_j)(t \approx s)$ is displayed in the form

n: [_u_1,...,_u_j] t = s

`write_disjunction_list` Writes a list of disjunctions of inequalities to the output stream.

`write_inequality_list` Writes a list of inequalities to the output stream.

`write_constraint_list` Writes a list of constraints to the output stream.

`write_cterm_list` Writes a list of constrained terms to the output stream.

`write_possibility_list` Writes a list of possible completion and normalization rule applications to the output stream.

5.14.5 Output of Complete

If the parameter *eqdebuglevel* is different from 0, `complete` displays information on the completion and normalization (the higher the value of *eqdebuglevel* the more). The information is sent to the current output stream.

As an example, we describe the information displayed if

$$\begin{aligned} E(A) &= \{(\forall x)(\forall y)(p(x,y) \approx p(x,x))\} \\ \mathcal{P}(A) &= \{\{\langle p(c,a), p(c,b)\rangle\}\} \ . \end{aligned}$$

First, the set of equalities $E(A)$ on the tableau branch and the set $\mathcal{P}(A)$ of E-unification problems to be solved are displayed. Quantifications such as $(\forall x)(\forall y)$ are represented by `[X,Y]`. The symbol `\=` means that this is an unification problem to be solved (only one in the example):

```
Beginning search for closure with equality of branch b1
Equalities extracted from Branch:
1: [X,Y]   p(X,Y) = p(X,X)
2: [X,Y]   p(X,X) = p(X,Y)

Problems extracted from Branch:
(1,1): p(c,a)  \=  p(c,b)
```

The initial rule system is displayed. Rules with inconsistent constraints have already been removed:

```
Initial System:
1: [X,Y] p(X,Y)=>p(X,X) << <[] and [gr(Y,X)]>
2: [X,Y] p(X,X)=>p(X,Y) << <[] and [gr(X,Y)]>
```

Now, the completion and normalization starts. "`0 left`" means that there is only one possible rules application (which is chosen); no other possibilities are left.

```
Chosen possibility (0 left):
critical_pair:
2: [X,Y] p(X,X)=>p(X,Y) << <[] and [gr(X,Y)]>
  --- 2: [X,Y] p(X,X)=>p(X,Y) << <[] and [gr(X,Y)]>  at [1] --->
[X,Y,Z,U] p(Z,Y)=>p(Z,U) << <[] and [gr(Z,U),gr(Z,Y)]>
```

The above output means that the rule

$$\mathbf{r}_2 = (\forall x)(\forall y)(p(x,x) \rightarrow p(x,y) \ll \langle id, x \succ y\rangle)$$

can b applied to itself at position $\langle 1\rangle$.

```
Critical pair rule applies

New Rules:
3: [X,Y,Z] p(Y,X)=>p(Y,Z) << <[] and [gr(Y,Z),gr(Y,X),gr(X,Z)]>
4: [X,Y,Z] p(Y,Z)=>p(Y,X) << <[] and [gr(Y,Z),gr(Y,X),gr(Z,X)]>
```

The application of the critical pair rule results in to new rules, which are displayed. Their constraints have already been transformed into normal form.

```
3: [X,Y,Z] p(Y,X)=>p(Y,Z) << <[] and [gr(Y,Z),gr(Y,X),gr(X,Z)]>
  subsumed by
4: [X,Y,Z] p(Y,Z)=>p(Y,X) << <[] and [gr(Y,Z),gr(Y,X),gr(Z,X)]>
```

The first of the new rules is subsumed by the second and therefore removed. Thus, the new reduction system is:

```
System:
1: [X,Y] p(X,Y)=>p(X,X) << <[] and [gr(Y,X)]>
2: [X,Y] p(X,X)=>p(X,Y) << <[] and [gr(X,Y)]>
4: [X,Y,Z] p(Y,Z)=>p(Y,X) << <[] and [gr(Y,Z),gr(Y,X),gr(Z,X)]>
0 possibilities left
```

"`0 possibilities left`" shows, that the completion terminates at this point.

Next, a normalization step is executed. The new rule $\mathbf{r}_4$ is applied to the term $p(c, a)$. A "non-simplification rule" can either be the critical pair rule or the derivation rule (as in this case).

```
Chosen possibility (2 left):
non_simplification:
[1,1,l,0]: [] p(c,a) << <[] and []>
  --- 4: [X,Y,Z] p(Y,Z)=>p(Y,X) <<
                 <[] and [gr(Y,Z),gr(Y,X),gr(Z,X)]>  at [] --->
[X,Y,Z] p(c,X) << <[] and [gr(a,X),gr(c,X)]>

Non-simplification rule applies

New Terms:
[1,1,l,1]: [X] p(c,X) << <[] and [gr(a,X),gr(c,X)]>

Terms:
[1,1,l,0]: [] p(c,a) << <[] and []>
[1,1,l,1]: [X] p(c,X) << <[] and [gr(a,X),gr(c,X)]>
2 possibilities left
```

The label `[1,1,l,1]` of the new term (which is universal w.r.t. the variable x) means that this is normal form No. 1 of the term which is the left side of the first part of the first simultaneous unification problem.

```
Chosen possibility (2 left):
non_simplification:
[1,1,r,0]: [] p(c,b) << <[] and []>
  --- 4: [X,Y,Z] p(Y,Z)=>p(Y,X) <<
                 <[] and [gr(Y,Z),gr(Y,X),gr(Z,X)]>  at [] --->
[X,Y,Z] p(c,X) << <[] and [gr(b,X),gr(c,X)]>

Non-simplification rule applies

New Terms:
[1,1,r,1]: [X] p(c,X) << <[] and [gr(b,X),gr(c,X)]>

Terms:
[1,1,r,0]: [] p(c,b) << <[] and []>
[1,1,r,1]: [X] p(c,X) << <[] and [gr(b,X),gr(c,X)]>
2 possibilities left
```

After a second derivation, the unification problem can be solved, and the tableau branch can be closed. The constraints $\langle id, (a \succ x) \wedge (b \succ x) \wedge (c \succ x)\rangle$ and ϵ are found to define solutions of the simultaneous E-unification problem.

```
[1,1] closed with constraint(s)
      <[] and [gr(a,X),gr(b,X),gr(c,X)]>

by combination of the normal forms
[1,1,r,1]: [X] p(c,X) << <[] and [gr(b,X),gr(c,X)]>
```

```
[1,1,l,1]: [X] p(c,X) << <[] and [gr(a,X),gr(c,X)]>

[1,1] closed with constraint(s)
      <[] and []>

by combination of the normal forms
[1,1,r,1]: [X] p(c,X) << <[] and [gr(b,X),gr(c,X)]>
[1,1,l,0]: [] p(c,a) << <[] and []>
```

The empty constraint ϵ is a solution, because $(\forall x)(p(c,x) \ll \langle id, (b \succ x) \wedge (c \succ x)\rangle)$ is universal w.r.t. x, and it is, thus, not necessary to instantiate x with a.

The simpler constraint ϵ and the empty substitution satisfying it are chosen to close the branch.

```
Branch closed with instantiation []

-------------------- PROOF --------------------
```

5.15 The Compiler

5.15.1 Calling the Compiler

The compiler is used for transforming knowledge bases from their external into their internal representation[25]. It is called by the predicate `parse` which is part of the `interface.pl` module. `parse` succeeds if the compiler does not find any errors in the input file. In that case the output is written to the same directory where the input file is read from. Error messages are written to the standard error stream (usually the shell).

The name of the output file is composed from the name of the input file and an extension. This extension is defined in the `sysdep` module within the predicate `internal_reset`. To check, which extension is pre-defined, use the predicate `get_tcplus_extension/1`.

The name of the compiler directory must either be defined in module `sysdep` in the variable `compdir`, or contained in the environment variable `THREETAP_COMPILER_DIR`. To read the name of compiler directory, the predicate `get_compiler_directory/1` can be used.

5.15.2 Implementation Language

The compiler is implemented using the Unix tools Lex and Yacc (resp. Flex and Bison). The name of the executable file is `parser`. The additional functions necessary for checking the input and generating the output are written in C.

If you want to change the parts of the compiler concerning the operators of the input language read Section 9.4.

[25] See Chapter 3 for the external representation and Chapter 5.5 for the internal representation of knowledge bases.

5.15.3 Scanner.l

The file `scanner.l` contains the `main` function of the `parser` file. It is the lexicographical analyzer of the compiler. Part of it is written in the Lex (resp. Flex) input syntax, and part of it is written in C. `scanner.l` contains the scanning rules for recognizing keywords and keycharacters of the grammar (see Table 3.1).

If no errors are detected, the message

```
no errors detected - compiling knowledge base
```

is printed to the standard error stream and the program exits with status 0. Else the program exits with status 1 after calling the function `yyerror`.

For changing the syntax of knowledge bases it might is useful to see what keywords and keycharacters are recognized. For that, an executable file `debug` can be generated. Use the shell command

```
debug < knowledgebase
```

to get a list of the recognized keywords and keycharacters.

5.15.4 Grammar.y

`grammar.y` contains the parsing rules which are given in Table 3.1. They are written in the Yacc (resp. Bison) input syntax. For checking the input and composing the output some additional C code is used. The more complex C functions have been put into the extra file `output.c`.

5.15.5 Output.c, Output.h

In `output.c` the functions which are necessary for checking the input and generating the output are located. `output.h` is the headerfile for `output.c`. The global datastructures of the compiler are defined here. In addition, the global functions of the `output.c` module are declared in the `output.h` headerfile.

6 Utility Programs

6.1 Visualizing Proofs

All the output utilities supplied with $_3T^AP$ operate on a special file which is generated by $_3T^AP$ during tableau construction if tableau output is switched on. By default it is switched off for efficiency. The command

```
set_tableau_output( on ).
```

enables the tableau output mode. Its counterpart

```
set_tableau_output( off ).
```

disables tableau output. Now let us assume tableau output mode is enabled and some theorem has been proved using some of the `prove/n` predicates. You will find a new file named `tableau.out` in the current directory—which is the directory you started $_3T^AP$ from or the one you changed to via `cd/1`. Although this file is more readable than the protocol output it is not easy to understand for a human. To overcome this deficiency two utilities are supplied: moreTab and tabTEX.

Remark 6.1 *Please note that all characters in the* command names *moreTab and tabTEX are typed lower case!*

6.1.1 Navigate Through Tableaux Using moreTab

moreTab is similar to the UNIX `more` utility. You may navigate through your tableau in a similar way you move through any UNIX text file. The tableau is indented, so you can see the branching structure of the proof.[1] moreTab has several options to suppress useless information, e.g. the sort `top`, or to display additional information, e.g. Dewey numbers of the branches. To invoke moreTab to display the prove in the file `tableau.out` simply type

```
moretab tableau.out
```

in your shell. To get a list of available commands type the `?`-key. moreTab's options are discussed below.

6.1.2 Typesetting Tableaux Using tabTEX

The second utility, tabTEX, may be used to typeset the proofs found by $_3T^AP$. As moreTab it reads the tableau output file generated by $_3T^AP$, but tabTEX has to read one more file, which is called `declarations.pl`. It is placed in the directory where all the $_3T^AP$ sources live. There tabTEX

[1] Since dissolution effects a branch in a more complicated way than other tableau rules do, moreTab might fail to display the results of a dissolution rule application properly.

finds the TEX-symbols which are used to typeset the formulae in the tableau. Assume we are in the directory `problems`, which is one level below the directory which contains the $_3T^AP$ sources (and, most important, the file `declarations.pl`) and assume further, the file `tableau.out` is also placed here. Now type

```
tabtex -s -d../declarations.pl
```

This will produce a file named `tap.sty`, a so called LATEX style file. If you want to include any tableau generated by tabTEX in one of your LATEX documents you have to add `tap` to your document styles, e.g. `\documentstyle[tap]{article}`. The style file needs not to be generated before typesetting every proof, you must reproduce it only when the operators had been changed. For details see below. `-d../declarations.pl` tells tabTEX to search for the declarations file in the parent directory. If not specified otherwise tabTEX searches that file in the current directory.

```
tabtex tableau.out test.tex -d../declarations.pl
```

produces the file `test.tex` which may be included in your LATEX document and which contains the LATEX-code of the tableau. As in moreTab the tableau is indented to visualize the structure of the proof tree. There are more options which will be described below.

6.2 $_3T^AP$'s Commands to Generate Tableau Output

Only few commands (predicates) are necessary to generate tableau output. Most of them have been introduced in the previous section. By default—for efficiency reasons—no output is generated by $_3T^AP$. `set_tableau_output(on)` tells $_3T^AP$ to redirect the output of the prover to some file. `set_tableau_output(off)` switches the tableau output off again. If not specified otherwise the tableau output file is located in the current directory and named `tableau.out`. You may use the `set_tableauoutfile(`*filename*`)` predicate to specify a different location or another filename. The active setting may be obtained by `get_tableauoutfile(F)`, the name of the current output file is returned in the Prolog-variable `F`. There is nothing else to do to achieve tableau output from $_3T^AP$.

Remark 6.2 *Please note that the tableau output file will be overwritten if you do not specify a new filename between two proofs.*

Online information concerning the output is available, type

```
info( output ).
```

Another way to look at the tableau output filename is to call the `lookup/0` predicate.

6.3 The Overall Structure of the Output

The output of moreTab or tabTEX is organized in so called *nodes*. A node consists of one or more lines of text, e.g. the three lines

```
The following extension(s) have been added:
   {f}p(a)
   {t}p(b)
```

constitute one node. There are a nine different kind of nodes in the tableau displayed by moreTab or typeset by tabTEX. These are

Tableau for A headline for the proof. The only location for this kind of node is the beginning of the file.

Rule applied to This node indicates a rule application.

Extension added The extension which has been generated by the above rule application is contained in this node.

Axiom added This node says that an axiom has been added to the tableau.

Branching point The previous rule generated at least two extensions. Thus the tableau branches here.

New subbranch This node indicates the beginning of a new subbranch.

Branch closed A subbranch is closed here.

Branch closed with equality Dito, but the closure has been achieved using equality reasoning.

Backtracking The current subbranch is exhausted and backtracking has been initiated.

6.4 moreTab

Here is a more detailed description of moreTab's features. In the first subsection the options (switches) are discussed. The second subsection deals with the available commands. An overview is given by two tables, one for the options the other for the commands.

6.4.1 Options

You may call moreTab from your shell by

moretab *filename options*

where *filename* is the name of the input file, i.e. the name you specified in $_3T^AP$ using the predicate `set_tableauoutfile(` *filename* `)` or the default filename `tableau.out`. *Options* is a sequence of some of the following available options:

Output dimensions To set the output height use the `-h`*n* switch where n is the number of output lines. In a similar way `-w`*m* sets the output width to m columns.

indentation `-i`*n* Sets the indentation factor to n characters. A subbranch is indented by n characters w.r.t. its parent branch.

Dewey numbers Use `+b` to enable the output of the Dewey numbers, i.e. the paths to the formulae in the branches. `-b` disables this option. It is disabled by default.

Top-sort `+t` causes moreTab to display the sort `top`. If `-t` is specified this is suppressed. The latter is the default behaviour.

Scroll portions By default the scroll portion is half a screenful of text. `+s` changes that to one screenful. `-s` specifies a scroll portion of half a screenful again.

Marks By default ten marks are available. With `-m`*n* you may set the maximum number of marks to any other value.

Version `-v` prints the version number of moreTab and the last changes to the program.

Batch mode The `-o`*filename* option switches moreTab in non interactive mode, i.e. the indented tableau is written to the specified output file instead to the standard output (your terminal by default). The flags and the indentation factor have their usual effects.

Table 6.1 contains an overview of available options.

switch	action	switch	action
`-w`*n*	set output width to n columns	`-h`*n*	set output height to n lines
`-i`*n*	set indentation factor to n	`-m`*n*	use n marks
`+b`	display the Dewey numbers	`-b`	don't display Dewey numbers
`+s`	scroll by one screenfuls	`-s`	scroll by half screenful
`+t`	print `top` sort info	`-t`	suppress `top` sort
`-v`	version number	`-o`*file*	batch mode

Table 6.1: moreTab options (n integer).

A call of moreTab without any argument will print out a short description of the options and the defaults.

6.4.2 Commands

moreTab knows five types of commands: commands for navigating through the tableau, commands associated with marks which may be used to remember special lines in the proof, commands to change flags, commands to customize the output and miscellaneous commands.

Navigation Commands

Moving up and down by lines

Type `n` to proceed one line (**n**ext line) and `p` to go back one line (**p**revious line). Please note that the lines are counted w.r.t. to the nodes which have been described in Section 6.3. I.e. the two lines of output

```
Tableau for
{f} p(a)
```

are counted as one (logical) line because they belong to the same node. Nodes are never split when displayed, therefore nothing will change if the above two lines of text are the last visible lines and you type `n`, but the next `n` will show the next node.

Moving up and down by pages

One page is a screenful or half a screenful of lines (nodes) where a screenful is the number of lines specified with the `-h` option or the `h` command. Whether one resp. a half screenful of text is scrolled is controlled by the `+s` resp. `-s` option or command as described in Section 6.4.1.

To scroll **f**orward use the `f` command or the `space` bar (as with UNIX `more`). To scroll `back` type `b`.

Moving up and down in the proof tree

The last method of navigating through the tableau is tree oriented. The `u` command brings you **u**p to the next branching point in the tableau which is above your current position. The `d` command moves you down to the next branching point below, i.e. the next leftmost

branching point of the tableau. `l` moves one branch to the **l**eft and `r` one branch to the **r**ight if possible, i.e. if there are any branches to the left or to the right.
Remark: If your current position is near the end of your file, moving up or left is relatively expensive on machines with poor performance.

Jumping to special positions
The percentage of text above the visible screenfull is displayed in the status line. It is possible to jump to a position with $p\%$ of text above, simply type the number p and the `%`-sign.
To move to the nth line type the number n directly followed by `g`.

Marks

moreTab manages ten marks, which may be set to lines you want to remember. As described in Section 6.4.1 the maximum number of marks may be changed. By n`m` mark number n is set to the first line in the display. Type n`#` to return to that position.

Flags

There are three flags which may be set or cleared by the following commands.

Dewey numbers `+b` activates the Dewey numbering and `-b` suppresses the display of Dewey numbers. By default Dewey numbers are not printed.

`top` sort If you are working in a domain with a single sort—whose sort is usually called `top` in $_3T^AP$—it may be useful to get rid of the sort information which is attached to every term. This is the default. The `-t` switch prevents the display of the `top` sort information. With `+t` you will get the `top` sort back.

Scroll portions The scroll commands scroll by one screenful of text or by half a screenful. The `+s` command tells moreTab to scroll by one screenful, this is the default behaviour. Use `-s` to scroll by half a screenful of text.

Remark 6.3 *Please note that numeric arguments have to appear* before *a command but* behind *a switch.*

Customize Output

moreTab will probably assume wrong dimensions of your shell window. To correct this, use the n`h` and m`w` commands. The former sets the output height to n lines while the latter sets the output width to m columns[2]. A better method to achieve correct output dimensions would be to use the appropriate switches with the current window geometry in an *alias* command of your shell. To adjust the indentation level of the proof subbranches to your personal style use the n`i` command. The command sets the indentation factor to n characters, i.e. each subbranch is indented n characters more than it's parent branch.

Miscellaneous

There are only two commands left. `?` shows the help page which contains a short command reference and `q` exits moreTab.

Table 6.2 shows an overview of the available commands for moreTab.

[2] Assuming a non-proportional font.

key	action	key	action
`n`	go to the next line	`p`	go to the previous line
`f` `space`	scroll forward	`b`	scroll backward

`d`
 	
`l`
	go down to
	leftmost next branching point
	move one branch to the left
		`u`
		
		`r`
			go up to
			previous branching point
			move one branch to the right

n%	move to the n% position	ng	go to line n
nm	set mark n	n#	return to mark n

`+b`
`+s`
`+t`
	display the Dewey numbers
	scroll by one screenfuls
	print `top` sort info
		`-b`
		`-s`
		`-t`
			don't display Dewey numbers
			scroll by half screenful
			suppress `top` sort

nh ni	set screen height to n lines set indentation factor to n characters	nw	set screen width to n columns
`q`	exit moreTab	`?`	Print the help page

A missing number is alway treated as 0.
If a command is prefixed by a number n it is executed n times.
(Of course this does not work for a command like `g` which starts with a number anyway.)

Table 6.2: moreTab commands (n integer).

6.5 tabTEX

In this section tabTEX is described in detail. The first subsection deals with the available options. In the second subsection the format of the style file is discussed and then an overview is given of how to customize the symbols used by LATEX for the logic's connectives.

6.5.1 Options

You may call tabTEX from your shell by

`tabtex` *tableau_filename LaTeX_filename options*

or

`tabtex -s` [*style_filename*] *options*

The second alternative will generate the style file while the first one typesets your tableau. *tableau-filename* is the name of the input file, i.e. the name you specified in $_3T^AP$ by

`set_tableauoutfile(` *filename* `)`

LaTeX-filename is the name of the output file generated by tabTEX. *Options* is a sequence of some of the following available options:

Declarations By default tabTEX searches the current directory for the file `declarations.pl`. It is not very likely that this file is in your working directory since `declarations.pl` is a part of the $_3T^AP$-source. The `-d` option tells tabTEX where to look for the declarations file. The path specification must directly follow `-d`.
Remark: tabTEX needs the declarations file to get the operators used by $_3T^AP$ and to read the default symbols for use with LATEX.

Style file If you want to include a tableau constructed by $_3T^AP$ into one of your LaTeXdocuments then you must add the $_3T^AP$style to your `\documentstyle`options list, an example is given in Section 6.1. To generate the style file use the `-s` option. The filename is optional, if specified it must directly follow the `-s`. If the filename is missing the style file is named `tap.sty`.
Remark: The extension of your filename should be `.sty` to make LaTeX happy.

Indentation As in moreTab, each subbranch of the tableau is indented. By default the indentation factor, i.e., the amount of space the subbranch is indented w.r.t. to it's parent, is 10mm. You may set the indentation factor to any valid LaTeXmeasure by simply adding `-i`*measure*, e.g. `-i5ex` will set the indentation factor to 15ex[3].

Dewey numbers As with moreTab you may add the Dewey numbers to the branches of your tableau. This is done by the `-b` option.

Top-sort Again as with moreTab you may suppress the `top` sort information which may be useless. Use the `-t` switch for that purpose.

Symbols To get a list of available operators and their LaTeX equivalents call tabTeX with the `-o` option. This will print such a list to your standard output.

If you call tabTeX without any argument a short usage information is printed. Table 6.3 contains an overview of the available options.

switch	action
`-d`*filename*	set path to declarations file
`-s`	generate style file named `tap.sty`
`-s`*filename*	generate style file with the specified name
`-i`*measure*	set indentation factor to *measure*
`-b`	print Dewey numbers
`-t`	suppress the `top` sort
`-o`	print the operator symbols for $_3T^AP$ and LaTeX

Table 6.3: tabTeX options (*filename* is any valid UNIX filename).

6.5.2 The Style File

The style file which must be included as a document style option contains definitions for several macros and some length-registers. Here is the style file generated by tabTeX using the `declarations` file from the two-valued version of $_3T^AP$:

```
\typeout{Document Style 'tap'. Generated by tabtex.}
\newcommand{\TAPOPdis }{\mbox{$\vee$}}
\newcommand{\TAPOPcon }{\mbox{$\wedge$}}
\newcommand{\TAPOPsneg }{\mbox{$-$}}
\newcommand{\TAPOPimp }{\mbox{$\supset$}}
\newcommand{\TAPOPequi }{\mbox{$\leftrightarrow$}}
\newcommand{\TAPOPall }{\mbox{$\forall$}}
\newcommand{\TAPOPex }{\mbox{$\exists$}}
```

[3] One *ex* is the height of the character *x* in the active font.

```
\newcommand{\TAPOPpmi }{\mbox{$\subset$}}
\newlength{\TAPindent}
\newlength{\TAPrest}
\newlength{\TAPparbackskip}
```

The first line is simply an identification which is printed whenever your LaTeX file is translated. The last three lines declare registers for lengths, these are used internally for the indentation of subbranches. The remaining lines of the style file introduce macros. There is one macro for every operator of the logic. The macro's name is `\TAPOP` followed by the internal name of the operator, e.g. `\TAPOPdis` for the disjunction. The macro's body is a mbox containing a math environment containing a backslash followed by the name specified through the `get_LaTeX_op_list/1` predicate in the `declarations` module. In the above example this is `\mbox{$\vee$}`, where `vee` is the name given by `get_LaTeX_op_list/1` which corresponds to `dis`.

Using the above mechanism it is not possible to create operator names like $\vee^3$ for the LaTeX output, since Prolog names must not contain the cărĕt-sign '^'. Nevertheless such names are possible. For the above example simply replace the `vee` in the `get_LaTeX_op_list/1` by `my_macro` and add a definition like

```
\newcommand{\my_macro }{\mbox{$\vee^3$}}
```

to your LaTeX file.

6.6 Syntax of moreTab's or tabTEX's Input

moreTab and tabTEX make the following assumptions concerning the input file's format:

1. Every line starts with a keyword (see below) except lines containing parts of an extension enclosed by '<' and '>'.
2. There is never a *newline*-character '`\n`' in any formula.
3. Every CLOSED-node is directly followed by a PROCEEDING-node or the end of file.

The syntax for the various kinds of nodes is given in Table 6.4. There, n is some integer, *path* is a list of integers separated by commata and enclosed in square brackets, e.g. [1,2,3], this is the notation used for Dewey numbers. *sign* and *formula* are treated as text and simply echoed by moreTab or tabTEX. Anything within square brackets is optional and $(a)^*$ says that a may be repeated any number of times. Texts in quotes stand for themselves.

‘TABLEAU FOR’ sign formula ‘`\n`’
‘RULE APPLICATION’ sign formula ‘TO’ branch ‘`\n`’
‘ADD EXTENSION `< \n`’ (formula ‘`\n`’)* formula ‘`>\n`’
‘PROCEEDING’ branch ‘REMAINING SUBBRANCHES (incl.)’ n ‘`\n`’
‘FORMULA(E)’ sign formula [‘AND’ sign formula] ‘`\n`’
‘CLOSED’ branch ‘`\n`’
‘EQUALITY CLOSURE BY AN INEQUALITY `\n`’ path ‘`\n`’
‘EQUALITY CLOSURE WITH’ sign formula [‘AND’ sign formula] ‘`\n`’ path ‘`\n`’
‘BACKTRACKING `\n`’

Table 6.4: moreTab’s and tabTEX’s input syntax

7 Getting Started — A Tutorial

7.1 Preliminary Remarks

This chapter tries to introduce you to ${}_3T^AP$ via a sample session. All the important concepts of ${}_3T^AP$ and the various settings and kinds of proofs will be mentioned.

We suppose that you have a ready installed version of ${}_3T^AP$ including all sources and some sample problems (see Appendix C). In the next section we describe the directory structure recommended for the installation of ${}_3T^AP$. Section 7.3 describes a typical session with ${}_3T^AP$ and introduces the various settings and features which are available. See also Section C.2.

7.2 Directory Structure

We recommend the following directory structure on your station and refer during the following to this structure.

1. The main directory for the installation of ${}_3T^AP$ is called `tap`. There are several subdirectories:
2. A subdirectory `Compiler` which contains the newest version of the compiler and related modules.
3. A subdirectory `2version` which contains the various modules of ${}_3T^AP$.
4. A subdirectory of `/tap/2version` called `problems` which contains the various problem files to be proven.

7.3 A Sample Session

We go to the `tap/2version/` directory and invoke Prolog. At the Prolog prompt we type in:

```
| ?- compile( boot ).
```

Now it will take several minutes (depending on your machine) to compile the various modules and the Prolog library predicates. After the compiling is finished, an information page is shown. If you are interested in some topics, e.g. information about the possible unix commands, type

```
| ?- info( unix ).
```

and the available information is shown. The available information pages are listed in Section 5.1.3.

Now we want to prove a problem. We choose Pelletier's 24th problem, which looks as follows:

axiom 1: $\neg(\exists x(s(x) \wedge q(x)))$
axiom 2: $(\forall x(p(x) \rightarrow (q(x) \vee r(x))))$
axiom 3: $(-(\exists x(p(x))) \rightarrow (\exists y(q(y))))$
axiom 4: $(\forall x((q(x) \vee r(x)) \rightarrow s(x)))$
theorem: $(\exists x(p(x) \wedge r(x)))$

We assume that we already have encoded it into the input syntax for the compiler and that it is stored in `tap/2version/problems` under the name `pel24`.

The problem in the correct input syntax for the compiler looks as follows:

```
sort top.

predicate p : top.
predicate q : top.
predicate r : top.
predicate s : top.

axiom pel24_1; -(exists x:top (s(x) & q(x))).
axiom pel24_2; (forall x:top (p(x) => (q(x) v r(x)))).
axiom pel24_3; (-(exists x:top (p(x))) => (exists y:top (q(y)))).
axiom pel24_4; (forall x:top ((q(x) v r(x)) => s(x))).

theorem pel24; (exists x:top (p(x) & r(x))).
```

Now we type at the Prolog prompt the following command:

```
| ?- cd( 'problems' ).
```

to change into the subdirectory `problems`.[1]

We now want to make a knowledge base (KB) from our problem `pel24` for the use with $_3T^AP$. There are two possibilities:

1. We create only the KB corresponding to our problem, this is done via

   ```
   | ?- compkbx( pel24 ).
   ```

   and read it afterwards into the workspace of Prolog via

   ```
   | ?- readkbx( pel24 ).
   ```

As you probably noticed, you can use the same file name both for your encoded problem and for the generated KB. $_3T^AP$ recognizes the "correct" file by the file name extension.

The output looks as follows:

```
no errors detected - compiling knowledge base
Reading compiler output : pel24.kb into pel24
Asserting               : pel24
Generating indices for  : pel24
Writing knowledgebase   : pel24    into pel24.kbx
```

[1] You can omit the quotes in the command above if the argument contains only letters and numbers and does not start with a number.

```
Deleting knowledgebase  : pel24
Reading knowledgebase   : pel24

The following theorems are defined:  [pel24]

yes
| ?-
```

The straightforward of way of proving `pel24` now is to type

```
| ?- prove( pel24 ).
```

and you will receive the following result:

```
Evaluation took 0.3 sec.
41 tableau rule applications
0 equality applications.
17 branches have been closed.
Backtracking has been tried 0 times.

-------------------- PROOF --------------------

yes
| ?-
```

If you are interested in a detailed protocol of the proof, you can do the following:

1. Select another output stream than screen

   ```
   | ?- set_tableau_output( on ).
   ```

2. If you do not want to send the output to the default output file `tableau.out`, then type for example

   ```
   | ?- set_tableauoutfile( 'pel24.out' ).
   ```

3. Start the proof again:

   ```
   | ?- prove( pel24 ).
   ```

4. Now you have a file `pel24.out` which you can examine in peace, for example via the `moretab` utility program. See Chapter 6 for a detailed description of the possibility to visualize proofs.

To continue our sample session, type now

```
| ?- set_tableau_output( off ).
```

to redirect the output stream to the screen.

If you are interested in some details about the proof, but you do not want to create an extra file, you can increase the numeric value of the global variable `debuglevel` to see more information on the proof derivation. Type for example

```
| ?- set_debuglevel( 2 ).
```

and start the proof again and look what happens. If this still does not satisfy your curiosity, you can choose a number between 1 and 5 for `debuglevel` to regain more and more extensive information.

Normally, the information which is shown when `debuglevel` is greater than 1 rushes so quickly over the screen that we recommend to redirect the output stream rather than to get the information via the debuglevel. Another possibility is to start $_3T^AP$ in an Emacs window running the Prolog shell.

If you have played a bit around with `debuglevel`, type

```
| ?- set_debuglevel( 0 ).
```

and look at further possibilities of $_3T^AP$.

Perhaps you have the feeling that you can achieve a more efficient proof of the same problem, or your own problem is even not provable with the default settings. Then you can use a different prove predicate or you can change various settings of the prover, for a complete description see Section 5.1.5 and Appendixes A and B.

First we show a different possibility to prove your problem. Instead of using the `prove/1` predicate we can also use `proveinc/1`. Type

```
| ?- proveinc( pel24 ).
```

This tries to find a proof with a successively higher bound on the number of γ rule applications starting with 1 (corresponding to `set_maxcounter(0)`). It stops as soon as a proof has been found on some level. With our example, there is no difference between the two possibilities, but there are examples where you can get much faster proofs (because of avoiding extensive backtracking) by using the `proveinc/*` predicates.

From the various possibilities to change global variables, we will demonstrate here only one; type

```
| ?- set_uselemmata( off ).
```

and afterwards

```
| ?- prove( pel24 ).
```

and you will receive the following result:

```
Evaluation took 0.3 sec.
51 tableau rule applications
0 equality applications.
22 branches have been closed.
Backtracking has been tried 0 times.

-------------------- PROOF --------------------

yes
| ?-
```

As you can see, the number of closed branches is greater than before. Therefore, the value `alpha` (the default) for `uselemmata` is the better choice.

If you type

```
| ?- set_uselemmata( on ).
```

and afterwards

```
| ?- prove( pel24 ).
```

you will receive the same result as with *uselemmata* set to `alpha`. You have to play around with these settings.

If you are done with a special problem, you can either delete it from the workspace via

```
| ?- delkb( pel24 ).
```

or you can just load another problem, say `pel41`. Type

```
| ?- usekbx( pel41 ).
```

and play around with it in the same manner as with `pel24`.

If you have not deleted `pel24` from the workspace, you can again make it the current knowledge base via

```
| ?- readkbx( pel24 ).
```

As you can see, it is possible to maintain several problems simultaneously in the workspace and flip from one to another.[2]

If you are not sure which the current knowledge base is or even which knowledge bases are in the workspace, you can use the `lookup/0` predicate to gain this information.

Please remember that besides the additionally available $_3T^AP$ predicates you are working in a standard Prolog shell. In particular, it is possible to exit the $_3T^AP$ system, for instance, by typing

```
| ?- halt.
```

We hope that this tutorial will help you to get a quick access to the famous $_3T^AP$ system and that you have as much fun with it as we did.

[2] If you change something in your input file, you must reload it in any case.

8 Evaluation

8.1 Problem Sets for Testing $_3T^AP$

8.1.1 The Statistical Information

In this section the problem sets are described, that have been used for testing and evaluating $_3T^AP$.[1]

For each problem the statistical information includes

- the name of the knowledge base containing the problem (column KB);
- the name of the proved theorem[2] (Theorem);
- the time in seconds $_3T^AP$ needs to find a proof running on a SUN-4 SPARC SLC work station (Time);
- the number of tableau rule applications[3] (RA);
- the number of equality applications to terms[4] (ERA);
- the length of the longest branch[3] (LBR);
- the number of closed branches[3] (CB);
- the number of backtracking occurrences, i.e., of parts of the proof being dismissed, (BT);
- the value of the parameter *maxcounter* used for the proof[5] (MC);
- the value of the parameter *maxbranchlength* used for the proof (MBR);
- the commands used to assign the switches and parameters values differing from their default values (Settings).

If no other settings are explicitly listed in the statistics, the switches and parameters have been set to their default values[6] (cf. Appendix B).

[1] These are the predefined problem sets that can be proved automatically using `proveall` (cf. Appendix A). The statistical information in this chapter has been generated by the command `proveall(all,tex)`.

[2] `none` if the inconsistency of the set of axioms is proved.

[3] Including those that are not part of the proof found, because they have been dismissed when backtracking occurred.

[4] Including those that lead nowhere or that are not part of the proof found, because they have been dismissed when backtracking occurred.

[5] That is the smallest value of *maxcounter* for which a proof can be found, unless it has been assigned to *maxcounter* by one of the commands listed in the column "Settings".

[6] However, *equality* is switched off, unless the problem to be proved is formulated using equality.

8.1.2 Simple Test Problems

The problem set `tests` consists of simple problems that demonstrate the effect that some of $_3T^AP$'s features have on proofs: handling of equality, demodulators, removing unlinked formulae, and universal formulae. The problems as well as their proofs are easy to understand and should therefore be consulted to get an idea of how these features work.

For statistical information see Table 8.1.

tests										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
`eqtest`	`t1`	0.017	2	1	4	1	0	0	0	equality: on
`eqtest`	`t2`	0.083	2	31	3	1	0	0	0	equality: on
`eqtest`	`t3`	0.050	6	2	7	2	0	0	0	equality: on
`eqtest`	`eq1`	0.033	1	0	2	1	0	0	0	equality: on
`eqtest`	`eq2`	0.050	2	1	4	1	0	0	0	equality: on
`eqtest`	`eq3`	0.034	3	2	5	1	0	0	0	equality: on
`demod`	`demod_1`	0.166	9	0	10	1	0	0	0	equality: on
`demod`	`demod_2`	0.467	10	0	11	1	0	0	0	equality: on
`demod`	`demod_3`	0.150	11	0	12	1	0	0	0	equality: on
`nested_terms`	`t1`	0.383	12	0	10	16	11	1	0	
`removetest`	`t`	0.033	6	0	6	2	0	0	0	removeunlinked: on, uselemmata: on, equality: on
`removetest`	`t`	0.050	9	0	7	3	0	0	0	removeunlinked: off, uselemmata: on, equality: on
`univ`	`t1`	0.033	4	0	6	3	0	0	0	equality: on
`univ`	`t2`	0.117	17	0	9	9	0	1	0	equality: on

Table 8.1: Statistics for the problem set `tests`.

8.1.3 D'Agostino's Problems

The problem class given by D'Agostino ((D'Agostino, 1990), page 69) is a sequence of unsatisfiable propositional formulae A_n $(n \geq 1)$, where A_n is the conjunction of the 2^n different clauses in

$$\{(L_1 \vee \ldots \vee L_n)\ :\ L_i = p_i \text{ or } L_i = \neg p_i,\ i = 1, \ldots, n\}.$$

The formulae A_n are of length $k = O(n2^n)$. D'Agostino claims that tableau proofs for the A_n are of length $O(2^k)$.

If the mechanism of lemma generation is used, the complexity of the proofs goes down to $O((n-1)2^n) = O(k \log k)$, because there is a closed tableau for A_n $(n \geq 1)$ with $1 + (n-1)2^n$ closed branches.

If *uselemmata* is switched on, the proofs generated by $_3T^AP$ are of approximately twice the length of these shortest proofs. Without lemma generation only A_2 and A_3 can be proved (to prove A_4 about 30,000 branches would have to be closed).

For statistical information see Table 8.2.

dagostino										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
dagostino2	none	0.050	9	0	7	5	0	0	0	uselemmata: on
dagostino2	none	0.067	11	0	8	7	0	0	0	uselemmata: off
dagostino3	none	0.183	43	0	17	21	0	0	0	uselemmata: on
dagostino3	none	0.867	149	0	23	101	0	0	0	uselemmata: off
dagostino4	none	0.666	148	0	37	73	0	0	0	uselemmata: on
dagostino5	none	3.050	450	0	77	225	0	0	0	uselemmata: on
dagostino6	none	13.416	1266	0	157	641	0	0	0	uselemmata: on

Table 8.2: Statistics for the problem set `dagostino`.

8.1.4 Murray & Rosenthal's Problems

The problems MR_n $(n \geq 1)$ proposed by Murray and Rosenthal (Murray & Rosenthal, 1987) are of the form

$$\begin{gathered}(p_{11} \vee \ldots \vee p_{1n}) \wedge \ldots \wedge (p_{n1} \vee \ldots \vee p_{nn}) \\ \wedge \\ ((\neg p_{11} \wedge \ldots \wedge \neg p_{1n}) \vee \ldots \vee (\neg p_{n1} \wedge \ldots \wedge \neg p_{nn})).\end{gathered}$$

The problems are of length $k = O(n^2)$. Murray and Rosenthal claimed the complexity of tableau proofs for the unsatisfiability of the MR_n $(n \geq 1)$ to be exponential in n, but this result holds only for Murray and Rosenthal's definition of semantic tableaux, that differs from ours. In the meantime, they do no more use this particular class of formulae as a counter example.

Using our definition for semantic tableaux one can find tableau proofs for MR_n with $n^2 = O(k)$ closed branches. $_3T^AP$ finds these short proofs if (and only if) *grepall* is switched off.

For statistical information see Table 8.3.

mr										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
mr2	mr2	0.100	10	0	8	4	0	0	0	grepall: off
mr3	mr3	0.100	24	0	12	9	0	0	0	grepall: off
mr4	mr4	0.150	44	0	16	16	0	0	0	grepall: off
mr5	mr5	0.250	70	0	20	25	0	0	0	grepall: off

Table 8.3: Statistics for the problem set `mr`.

8.1.5 Cook & Reckhow's Problems

The problems CR_n $(n \geq 1)$ given by Cook and Reckhow (Cook & Reckhow, 1974) are conjunctions of the 2^n clauses

$$L^1 \vee L^2_{i_2} \vee \ldots \vee L^n_{i_n},$$

where $L^j_{i_j}$ is either $p^j_{i_j}$ or $\neg p^j_{i_j}$ and the Index i_j is the sequence of the signs of the $L^1, \ldots, L^{j-1}_{i_{j-1}}$. These conjunctions are to be proved to be unsatisfiable.

Cook and Reckhow's problems are in some way very similar to D'Agostino's. They, too, are of length $k = O(2^n)$, the complexity of their tableau proofs is $O(2^k)$, and if lemma generation is used, much shorter tableau proofs with $1 + (n-1)2^n$ closed branches and complexity $O(k \log k)$ can be found.

For statistical information see Table 8.4.

cr										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
`cr2`	`none`	0.067	10	0	7	5	0	0	0	uselemmata: on
`cr2`	`none`	0.067	13	0	6	7	0	0	0	uselemmata: off
`cr3`	`none`	0.217	56	0	17	29	0	0	0	uselemmata: on
`cr3`	`none`	0.800	247	0	18	121	0	0	0	uselemmata: off
`cr4`	`none`	3.567	601	0	37	302	0	0	0	uselemmata: on

Table 8.4: Statistics for the problem set `cr`.

8.1.6 Kalish & Montague's Problems

The problem set `kalish` consists of 22 problems taken from (Kalish & Montague, 1964). The problems have been numbered in the same way as in that book, e.g. `kalish201` is problem *T201* in (Kalish & Montague, 1964).

All of the `kalish` problems consist of a single theorem and have no axioms. Most of them are very easy to prove—except for `kalish265`, which cannot be proved by $_3T^AP$. `kalish317` uses equality and can only be proved with *maxcounter* set to 1.

For statistical information see Table 8.5.

kalish										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
`kalish201`	`kalish201`	0.116	6	0	8	2	0	0	0	
`kalish202`	`kalish202`	0.100	6	0	8	2	0	0	0	
`kalish203`	`kalish203`	0.100	9	0	6	2	0	0	0	
`kalish204`	`kalish204`	0.133	9	0	6	2	0	0	0	
`kalish215`	`kalish215`	0.116	9	0	7	4	0	0	0	
`kalish223`	`kalish223`	0.117	7	0	6	4	0	0	0	
`kalish227`	`kalish227`	0.084	3	0	3	2	0	0	0	
`kalish229`	`kalish229`	0.083	3	0	4	1	0	0	0	
`kalish230`	`kalish230`	0.100	3	0	4	1	0	0	0	
`kalish234`	`kalish234`	0.083	6	0	8	3	0	0	0	
`kalish238`	`kalish238`	0.100	3	0	4	1	0	0	0	
`kalish239`	`kalish239`	0.100	6	0	8	2	0	0	0	
`kalish240`	`kalish240`	0.100	8	0	10	3	0	0	0	
`kalish241`	`kalish241`	0.100	9	0	11	4	0	0	0	
`kalish244`	`kalish244`	0.100	5	0	7	1	0	0	0	
`kalish246`	`kalish246`	0.134	9	0	7	3	0	0	0	
`kalish249`	`kalish249`	0.100	5	0	6	1	0	0	0	
`kalish250`	`kalish250`	0.117	17	0	10	2	0	0	0	
`kalish255`	`kalish255`	0.100	6	0	6	2	0	0	0	
`kalish256`	`kalish256`	0.184	17	0	11	7	4	0	0	
`kalish317`	`kalish317`	0.433	51	5	26	20	0	1	0	equality: on, maxcounter: 1

Table 8.5: Statistics for the problem set `kalish`.

8.1.7 Problems Constructed according to Morgan

These four problems are constructed by means of Morgan's method (Morgan, 1976) of encoding problems from propositional logic in predicate logic. By encoding very simple tautologies from propositional logic, first-order problems have been generated that are very difficult to prove. For statistical information see Table 8.6.

meta_pl										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
`meta_pl`	`t1`	4.317	24	0	23	28	19	3	0	maxcounter: 3, equality: on

Table 8.6: Statistics for the problem set `meta_pl`.

8.1.8 The "Pigeonhole" Problems

The pigeonhole problems are propositional theorems defined for each $n \geq 1$. In English:

> There are $n+1$ pigeons and n pigeonholes. Each pigeon is in one hole. Therefore, it is not possible, that in each hole there is exactly one pigeon.

There are two different formulations of this problem in propositional logic. The first one (problem set `pigeon`) is the conjunction of the $n+1$ disjunctions

$$p_{i1} \vee \ldots \vee p_{in}, \; 1 \leq i \leq n+1$$

implies the disjunction of the $\frac{1}{2}(n^3+n^2)$ conjunctions

$$p_{ik} \wedge p_{jk}, \; 1 \leq i < j \leq n+1, \; 1 \leq k \leq n.$$

The second formulation (problem set `pig_alt`) is that the conjunction of the $\frac{1}{2}(n^3+n^2)$ clauses

$$p_{ik} \supset \neg p_{jk}, \; 1 \leq i < j \leq n+1, \; 1 \leq k \leq n.$$

and the $n+1$ clauses

$$p_{i1} \vee \ldots \vee p_{in}, \; 1 \leq i \leq n+1$$

which has to be unsatisfiable.

The second formulation is much more suitable for proving the "pigeonhole" problems using $_3T^AP$. The reason is, yet, unknown.

Both formulations are of length $k = O(n^3)$. In theory the length of the tableau proofs grows exponentially in k. Probably, the complexity cannot be reduced by using lemma generation. This assumption is enforced by the statistics, although there is no proof yet.

For statistical information see Tables 8.7 and 8.8.

pigeon										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
`pigeon2`	`pig2`	0.333	47	0	19	40	0	0	0	uselemmata: on
`pigeon2`	`pig2`	1.000	175	0	18	168	0	0	0	uselemmata: off
`pigeon3`	`pig3`	9.500	749	0	49	714	0	0	0	uselemmata: on
`pigeon3`	`pig3`	76.300	629	0	70	102	0	0	0	uselemmata: on, dissolution: on

Table 8.7: Statistics for the problem set `pigeon`.

pig_alt										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
pig_alt2	none	0.183	21	0	11	14	0	0	0	uselemmata: on
pig_alt2	none	0.150	20	0	9	14	0	0	0	uselemmata: off
pig_alt3	none	1.183	162	0	27	90	0	0	0	uselemmata: on
pig_alt3	none	1.350	166	0	22	106	0	0	0	uselemmata: off
pig_alt4	none	9.717	949	0	48	536	0	0	0	uselemmata: on
pig_alt4	none	7.800	1088	0	37	692	0	0	0	uselemmata: off
pig_alt5	none	92.683	5697	0	73	3266	0	0	0	uselemmata: on
pig_alt5	none	67.117	7370	0	56	4730	0	0	0	uselemmata: off

Table 8.8: Statistics for the problem set pig_alt.

8.1.9 Problems from Group Theory

The following two problems from group theory are mainly for testing the handling of equality. The tableau proofs for both of them consist of one branch. All formulae are universal with respect to all free variables.

The first problem gr1 is to prove that from the axioms

$$\begin{array}{l}(\forall x)(e \cdot x \approx x)\\(\forall x)(x \cdot x^{-1} \approx e)\\(\forall x)(x^{-1} \cdot x \approx e)\\(\forall x)(\forall y)(\forall z)(x \cdot (y \cdot z) \approx (x \cdot y) \cdot z)\end{array}$$

the additional axiom

$$(\forall x)(x \cdot e \approx x)$$

can be derived. The second problem gr2 is to prove, now using the additional axiom as well, that

$$(\forall x)(\forall y)(\forall z)(x \cdot y \approx z \cdot y \supset x \approx z)$$

is a theorem from group theory.

For statistical information see Table 8.9.

groups										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
gr1	gr1	2.933	5	124	6	1	0	0	0	equality: on
gr2	gr2	146.034	7	1281	9	1	0	0	0	equality: on

Table 8.9: Statistics for the problems set groups.

8.1.10 Pelletier's Problems

In (Pelletier, 1986) Pelletier gives a collection of a lot of different problems for testing automatic theorem provers. This is the problem set most often used for testing $_3T^AP$. It has been divided into three subsets: problems from propositional logic (pel_prop), problems from predicate logic (pel_pred), and problems formulated with equality (pel_eq).

Pelletier used a scale from 1 (easiest) to 10 (most difficult) points to denote the difficulty of the problems (from his point of view). His judgement has been included in the statistics (Tables 8.10, 8.12 and 8.11).

Pelletier's 47th (Schubert's Steamroller) and 51st–54th problem could, yet, not be proved.

pel_prop											
KB	Theorem	Diff.	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
pel1	pel1	2	0.083	9	0	7	4	0	0	0	
pel2	pel2	2	0.083	5	0	4	2	0	0	0	
pel3	pel3	1	0.067	4	0	6	1	0	0	0	
pel4	pel4	2	0.083	9	0	6	4	0	0	0	
pel5	pel5	4	0.100	6	0	7	3	0	0	0	
pel6	pel6	2	0.083	2	0	3	1	0	0	0	
pel7	pel7	3	0.083	4	0	5	1	0	0	0	
pel8	pel8	5	0.083	3	0	5	2	0	0	0	
pel9	pel9	6	0.134	14	0	12	6	0	0	0	
pel9	pel9	6	0.366	7	0	9	1	0	0	0	dissolution: on
pel10	pel10	4	0.150	15	0	7	14	0	0	0	
pel11	pel11	1	0.083	1	0	3	2	0	0	0	
pel12	pel12	7	0.217	23	0	7	24	0	0	0	
pel13	pel13	5	0.150	14	0	8	9	0	0	0	
pel14	pel14	6	0.150	18	0	9	10	0	0	0	
pel15	pel15	5	0.100	7	0	6	4	0	0	0	
pel16	pel16	4	0.067	3	0	5	1	0	0	0	
pel17	pel17	6	0.167	28	0	14	12	0	0	0	
pel17	pel17	6	0.617	22	0	13	4	0	0	0	dissolution: on

Table 8.10: Statistics for the problem set pel_prop.

pel_eq											
KB	Theorem	Diff.	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
pel48	pel48	3	0.167	4	24	5	4	0	0	0	equality: on
pel49	pel49	5	0.367	21	92	11	10	2	2	0	equality: on, uselemmata: off, maxcounter: 2
pel50	pel50	4	0.117	6	0	7	2	0	0	0	equality: on
pel56	pel56	4	0.134	13	2	9	5	0	0	0	equality: on
pel57	pel57	2	0.117	3	0	5	3	0	1	0	equality: on, maxcounter: 1
pel58	pel58	3	0.100	2	2	3	1	0	0	0	equality: on
pel59	pel59	3	0.200	17	0	12	8	2	1	0	equality: on
pel60	pel60	4	0.116	10	0	8	4	0	0	0	equality: on
pel61	pel61	6	0.150	2	4	3	1	0	0	0	equality: on
pel62	pel62	5	0.150	17	0	13	6	0	0	0	equality: on

Table 8.11: Statistics for the problem set pel_eq.

8.1.11 Other Two-Valued Problems

The set ps is a class of very hard challenge problems proposed by P. H. Schmitt. The formulae PS_n are corollaries of the well known fact that an injective function $f : M \longrightarrow M$ that operates on a finite set M has to be surjective.

$$\frac{\begin{array}{c}(\forall x)(x \not\approx a_1 \wedge \ldots \wedge x \not\approx a_n \supset f(x) \approx x) \\ (\forall x)(\forall y)(f(x) \approx f(y) \supset x \approx y)\end{array}}{(\forall x)(\exists y)(x \approx f(y))}$$

The problems PS_n can be seen as a formulation of the "pigeonhole" problems using equality (if

pel_pred											
KB	Theorem	Diff.	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
pel18	pel18	1	0.100	3	0	5	1	0	0	0	
pel19	pel19	3	0.116	6	0	8	2	0	0	0	
pel20	pel20	4	0.133	14	0	15	3	0	0	0	
pel21	pel21	5	0.134	10	0	10	7	0	1	0	
pel22	pel22	3	0.100	7	0	6	4	0	0	0	
pel23	pel23	4	0.116	9	0	7	4	0	0	0	
pel24	pel24	6	0.267	41	0	17	17	0	0	0	
pel24	pel24	6	0.850	38	0	18	5	0	0	0	dissolution: on
pel25	pel25	7	0.167	18	0	16	7	0	0	0	
pel26	pel26	7	0.233	40	0	15	12	0	0	0	
pel27	pel27	6	0.350	57	0	15	22	7	0	0	uselemmata: off
pel28	pel28	8	0.167	20	0	14	5	0	0	0	
pel29	pel29	7	0.333	50	0	20	19	0	1	0	maxcounter: 1
pel30	pel30	6	0.117	10	0	9	3	0	0	0	
pel31	pel31	5	0.117	11	0	13	4	0	0	0	
pel32	pel32	6	0.166	16	0	14	9	0	0	0	
pel33	pel33	4	0.184	26	0	13	9	0	0	0	
pel34	pel34	10	1.233	219	0	22	76	0	1	0	maxcounter: 1
pel35	pel35	2	0.117	5	0	6	1	0	0	0	
pel36	pel36	3	0.133	13	0	11	3	0	0	0	uselemmata: off
pel37	pel37	3	0.183	23	0	19	5	0	0	0	
pel38	pel38	4	67.900	1343	0	46	1671	1011	1	0	uselemmata: off
pel39	pel39	3	0.133	6	0	6	2	0	0	0	
pel40	pel40	5	0.133	13	0	11	5	1	0	0	
pel41	pel41	6	0.150	14	0	10	6	3	0	0	
pel42	pel42	6	0.133	15	0	14	5	0	2	0	maxcounter: 2
pel43	pel43	5	1.833	165	0	31	135	38	1	0	uselemmata: off, maxcounter: 1
pel44	pel44	3	0.150	15	0	16	5	1	0	0	uselemmata: off
pel45	pel45	5	0.300	46	0	38	15	0	1	0	uselemmata: off, maxcounter: 1
pel45	pel45	5	0.817	33	0	23	4	0	1	0	uselemmata: off, dissolution: on, maxcounter: 1
pel46	pel46	6	0.200	26	0	20	10	0	4	0	uselemmata: off, maxcounter: 4

Table 8.12: Statistics for the problem set pel_pred.

one assumes f to be the function that assigns each pigeon a pigeonhole). $_3T^AP$ can only prove them for $n = 1$. For statistical information see Table 8.13.

ps										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
ps1	ps1	0.633	64	51	16	17	1	1	0	uselemmata: off, maxcounter: 1, equality: on

Table 8.13: Statistics for the problem set ps.

The formula Φ_4 (problem set `phi`) from (Murray & Rosenthal, 1993) is constructed in such a way that it demonstrates the advantages of dissolution (Table 8.14).

phi										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
phi4	phi4	21.333	6021	0	28	1024	0	0	0	
phi4	phi4	10.267	121	0	29	18	0	0	0	dissolution: on
phi4	phi4	3.800	27	0	22	4	0	0	0	dissolution: on, disscomplexity: on

Table 8.14: Statistics for the problem set phi.

8.1.12 Problems from Three-Valued Logic

The three-valued problems are taken from (Gerberding, 1991), where an axiomatization for natural interval arithmetic[7] using a three-valued logic which assembles parts of the Lukasiewicz logic Ł$_3$ and the three-valued Gödel system $\mathbf{G}_3$.

The problems are as follows:[8]

eq4 `eq4` states the symmetry of the three-valued equality predicate for intervals. It directly follows from `fig5_14` and `fig5_17` (see below). Although the axioms and the theorem of `eq4` are first-order formulae, the problem is of the form

$$\frac{\begin{array}{c} j_1 a \leftrightarrow j_1 b \\ j_{\frac{1}{2}} a \leftrightarrow j_{\frac{1}{2}} b \end{array}}{a \leftrightarrow b}$$

$_3T^AP$ is able to prove `eq4` from the axioms given in (Gerberding, 1991), without using `fig5_14` and `fig5_17` as lemmata.

l5_9 This theorem says that the two-valued equality of intervals, i.e., their set-theoretic equality, may be stated using the three-valued equality predicate eq.

fig5_14 `fig5_14` states that the above equality predicate is symmetrical w.r.t. the truth value $\frac{1}{2}$, i.e.,

$$j_{\frac{1}{2}} eq(a,b) \leftrightarrow j_{\frac{1}{2}} eq(b,a)$$

for any two intervals a, b.

fig5_17 `fig5_17` is similar to `fig5_14`. It states the symmetry of eq wrt the truth value 1, i.e.,

$$j_1 eq(a,b) \leftrightarrow j_1 eq(b,a)$$

for any two intervals a, b.

l5_1 This problem states the antisymmetry of the order relation defined in (Gerberding, 1991) wrt the truth value $\frac{1}{2}$.

th5_3 This is the antisymmetry of the above ordering.

For statistical information see Table 8.15. A more detailed discussion concerning the problems of proving these theorems with an automated theorem prover may be found in Chapter 7 of (Gerberding, 1991). More three-valued problems may be found there, too.

[7] This is the arithmetic of intervals with positive integer bounds. It is some kind of generalization of the Peano arithmetic for naturals.

[8] Names starting with `l` are lemmata from (Gerberding, 1991), e.g. `l5_1` is Lemma 5.1, names starting with `th` are theorems and names starting with `fig` are formulae whose proof may be found in a figure in (Gerberding, 1991). `eq4` is axiom MVEQ4.

three_valued										
KB	Theorem	Time[s]	RA	ERA	LBR	CB	BT	MC	MBR	Settings
eq4	eq4	0.583	94	0	23	28	0	0	0	
fig5_14	fig5_14	8.117	566	0	55	371	128	4	0	maxcounter: 4
fig5_17	fig5_17	0.250	58	0	25	8	0	1	0	grepall: off
15_1	15_1	0.250	46	0	17	25	0	0	0	
th5_3	th5_3	0.850	216	0	24	65	0	0	0	

Table 8.15: Statistics for the problems set three_valued.

8.2 Shortcomings and Strengths

Among the major merits of using tableaux as a logical basis for mechanical theorem proving are, first, that they do not commit one to the usage of normal forms; second, that, therefore, the proofs generated are relatively easy to understand; and, third, that they can be extended to cover many of the nonclassical logics. All these advantages apply to ${}_3T^AP$:

+ No normal form is required.

+ In contrary to other automated theorem provers (in particular resolution-based theorem provers) ${}_3T^AP$ generates proofs that are relatively easy to examine (using the tools moreTab and tabTEX).

+ An equality theory that may be part of a problem does not have to be specified explicitly. Equalities may occur arbitrarily nested in formulae.

+ ${}_3T^AP$ can handle every multiple-valued logic that can be defined by means of truth-tables. To adopt a new logic one has only to describe the operators and tableau rules of that logic. ${}_3T^AP$'s proof procedure does not have to be changed.

The main disadvantage of ${}_3T^AP$ in comparison to state-of-the-art resolution-based theorem provers for predicate logic is its

− lower performance.

The reason is mainly that ${}_3T^AP$ is implemented in Prolog. If one compares the number of inference steps, ${}_3T^AP$ can compete with state-of-the-art theorem provers. The fact that resolution-based theorem provers are in a better proof complexity class is in practice not harmful—due to lemma generation and other sophisticated tableau proving techniques ${}_3T^AP$ makes use of.

+ ${}_3T^AP$ is able to prove problems from propositional logic in a rather short time. It takes only between 0.04s and 0.27s[9] to prove the propositional problems given in (Pelletier, 1986). Propositional challenge problems (e.g. the Pigeonhole Problems) can be proved in relatively short time, too.

+ Certain problems can be proved very fast because no normal form has to be generated, e.g. Andrew's Challenge (Pelletier's 34th problem) can be proved in 2.4s[9].

± As long as no bracktracking occurs problems from predicate logic can as well be proved in rather short time. Most of Pelletier's problems from predicate logic can be proved in less than one second[9]. Difficulties arise if extensive backtracking is necessary, i.e., if there are

[9] On a SUN-4 SPARC SLC work station.

often several possibilities to close a branch leading to different free variable instantiations. This is the reason why, for example, Schubert's Steamroller (Pelletier's 47th problem) cannot be proved.

± Although performance cannot compete with completion based theorem provers or resolution systems using paramodulation, $_3T^AP$'s performance in proving problems with equality is approximately the same as in proving problems from predicate logic without equality. However, if too many equality applications are necessary to prove two terms to be equal, and if these applications are difficult to find because there are a lot of different equalities that can be applied to each term, $_3T^AP$ may fail to find a proof. $_3T^AP$ will, for example, only succeed in proving a theorem from group theory, if not more than about five equality applications are necessary to show two terms to be equal.

The advantages and disadvantages of $_3T^AP$'s user interface (especially of using the Prolog shell) are:

+ The user can quickly write his own predicates to accomplish specialized tasks (e.g., abbreviations for certain command sequences).

− Since the user interface is command line oriented, learning to use ist is not as easy as, for example, a window-based system.

+ It is very easy to include $_3T^AP$ in any existing system via its interface predicates.

+ Several knowledge bases can be loaded into the work space at the same time, and it is possible to switch between these knowledge bases.

+ $_3T^AP$ provides various switches and parameters having an influence on the prove procedure. Though it might sometimes be difficult to find the best settings for the switches, the default settings work very well in most cases.

8.3 Settings of Switches and Parameters

In this section we give some hints on how to use $_3T^AP$'s various switches and parameters. Some of these are easy to explain, others are merely empirical results.

8.3.1 Settings of Switches that Might Help if No Proof is Found

The following list contains settings of switches and parameters you should try if $_3T^AP$ fails to find a proof.

- Increase *maxcounter* or use the command `proveinc` if $_3T^AP$ fails to find a proof for a problem from predicate logic (*maxcounter* has no influence on propositional logic proofs).
- Set *maxbranchlength* to a higher value or to 0, or us the command `proveinc`.
- Set *maxcounter* to a very high value (e.g. 1000), such that no backtracking occurs, if you think too much backtracking is the reason why $_3T^AP$ doesn't find a proof.
- Set *max_solutions_per_branch* to a smaller value if backtracking occurs and the problem contains equalities.

- Switch *flipconclusion* on if the problem consists of asymmetric formulae, especially if it contains implications.
- Switch *grepall* off if the knowledge base contains a relatively large number of axioms (because it might be not necessary to put all axioms on all branches).
- Switch *uselemmata* off (that works for some problems from predicate logic).

8.3.2 Settings of Switches that Might Shorten the Proof Found

Of course, the settings given in the previous section may not only help to find a proof at all, but may also shorten the proof found. There are some other settings that usually lead to a shorter proof, though they might increase the time $_3T^AP$ needs to find it,

- Try to set *maxcounter* to s amaller value. Even if that means that backtracking occurs, the proof found will be shorter.
- Switch *removeunlinked* on.
- Use dissolution. To further decrease the proof length switch *disscomplexity* on.

9 Using Different Logics

9.1 $_3T^AP$'s Logics

The logic used by $_3T^AP$ is defined in the modules `declarations` and `rules`. In most cases it is sufficient to change these to add a new logic or change an old one. `declarations` has to be adjusted to the signature of the logic, and `rules` represents the logic's semantics[1]. Only to add new operators, or to change their names, arity, or priority, some files of the compiler have to be edited, in addition.

The standard distribution of $_3T^AP$ comes with the following pre-defined logics:

- The classical two-valued predicate logic (defined in the directory `2version`).
- The three-valued Łukasiewicz logic $Ł_3$ (defined in `3version/declarations_std.pl` and `3version/rules_std.pl`).
- The three-valued logic introduced in (Gerberding, 1991); it assembles parts of the Łukasiewicz logic $Ł_3$ and the Gödel system $\mathbf{G}_3$. Besides the definition in `declarations_sg.pl` and `rules_sg.pl`, the directory `3version` contains the Carnielli version of this logic (`declarations_sg_car.pl` and `rules_sg_car.pl`), where signs represent only a single truth value, and a version (`declarations_sg_as.pl` and `rules_sg_as.pl`), that uses all possible combinations of truth values signs[2].
- The seven-valued logic introduced in (Kernig, 1992) (defined in the directory `7version`).

To switch between the different three-valued logics, copy the `declarations` and `rules` modules defining the logic you want to use to `3version/declarations.pl` resp. `3version/rules.pl`. Then re-compile the prover (see Appendix C).

9.2 Changes to the Declarations Module

The following predicates have to be adjusted to the signature of your logic:

- `get_int_op_list`
- `get_ext_op_list`
- `get_LaTeX_op_list` (only necessary if the tabTEX utility is going to be used)
- `get_ops_with_two_fmas`
- `get_unary_ops`

[1] Some aspects of the semantics are represented in `declarations`, e.g. for any self-contradictory combination of a connective and a sign there is an appropriate fact in `declarations`.

[2] Note, that $_3T^AP$ is not complete for this versions of the logic.

- `get_int_quantor`
- `get_ext_quantor`

You cannot use arbitrary internal and external names for the connectives, but only those known to the compiler.

If a rule must be applied more than once to a signed formula in order to achieve completeness (cf. Sections 2.7, 5.9.4) then the formula is of type γ and a clause of the `is_gamma_formula/2` predicate has to be added for that formula, see Section 5.9.4.

If a sign σ does not appear in the truth table of a connective c then any signed formula $\sigma\ c(\cdots)$ is self-contradictory and there is no rule defined for that combination. Hence, the fact `no_rule_defined(` σ`,`c `)` must be present in `declarations`. See Section 5.9.3 for details.

For every pair of complementary signs a clause of the `is_complementary_sign/2` predicate has to be created. See Section 5.9.2 for details.

9.3 Changes to the Rules Module

After the tableau rules have been extracted from a connective's truth table it is quite easy to change the `rules` module. For every connective and for every sign a clause of the `rules/8` predicate must be supplied. The format of these clauses is described in Section 5.10. If no rule is defined for a pair of sign and operator a dummy rule has to be added, see Section 5.10.1.1.

If the universal formula mechanism (Section 2.5) is not to be used you should use a `rules/6` predicate instead of `rules/8` and the following interface clause:

```
rule( A,B,C,D,E,_,[],F ) :- rule( A,B,C,D,E,F ).
```

The `rules` module of the many-valued versions of $_3T^AP$ are written in that way, since they do not make use of the universal formula mechanism.[3]

To use the lemma generation mechanism some clauses must be created for the `get_lemmata/6` or `get_lemmata_alpha/6` predicates. Their format is discussed in Section 5.10.2. Additional information, may be gathered form the source of the `rules` module. For the many-valued case it is not easy, although well possible, to compute the required lemmata. See (Hähnle, 1992c) for some hints.

9.4 Changes to the Compiler

9.4.1 Defining New Operators

For adding operators there are three things to think about: The external and the internal representation, the priority and the arity of the new operator. Whether you only have to change the representation or whether you even have to change the priority or the arity depends on how your new operator differs from the so far given. The priority of the pre-defined operators for two-valued and three-valued logics is listed in Table 9.1 (all of these operators are unary or binary). There are several markers in the modules of the compiler which indicate the locations that must be edited. The next sections describe in detail how these locations must be modified.

[3] There is no principal reason speaking against it. However, in the many-valued case few rules have singleton conclusions and, therefore, universal formulae are not so frequent.

max. priority

`<=>`	EQUI

`#`
`>, < >, <`
`->, <->, <-`
`=>, <=`
	BINOP

`v`	DIS_OR_V
`&`	CON

`-,`
`aff, jf, ju, jt, nabla`
	UNOP

min. priority

Table 9.1: Priority classes of the operators.

9.4.2 External and Internal Representation

The first step is to decide on the external and internal representation of a new operator. Make sure that the external representation is not a single letter, a single number, or the character '_', and of course none of the keywords or keycharacters which are already used in the grammar (see Table 3.1).

The internal representation must be unique, too. See Tables 5.2 and 5.3 for what internal representations are already used.

If the new operator does not belong to one of the given priority classes (except the `DIS_OR_V` class), see Table 9.1, or if it is neither unary nor binary, then you have to add a new priority class (see section 9.4.3).

To add the new operator, you have to edit the file `scanner.l`. At the position marked by `CHANGE_REPRESENTATION_HERE` you will find the scanning rules for operators. Every line in the marked block contains the data for one operator. For example, the line

```
    "=>"    { yylval="imp";     return token(BINOP); }
```

defines the implication operator: `=>` is the entry for the external representation, `yylval="imp";` links the external to the internal representation `imp`, and `return token(BINOP);` defines the priority class of the implication operator.

Insert a line with the data for your new operator. If you have added a new priority class, you also have to change some locations concerning the priority of operators (see Section 9.4.3).

9.4.3 Priority

There are three locations in concerning the priority of operators. These locations are indicated by the marker `CHANGE_PRIORITY_HERE`.

One of these locations is in the file `scanner.l` at the list of scanning rules, where each operator is linked to a priority class (as already mentioned in Section 9.4.2). In Table 9.1 you can see the priority classes of the predefined operators.

To change the priorities, you also have to edit the file `grammar.y` (at the positions marked by `CHANGE_PRIORITY_HERE` marker). Here, you will find the definition of tokens for the operators in order of decreasing priority. All priority classes used in the file `scanner.l` must appear here. The order of the entries in the list defines the priority of the priority classes. Entries in the same line have the same priority.

The second location at which you find the `CHANGE_PRIORITY_HERE` marker in `grammar.y` are the parsing rules for operators. For a new priority class you have to add a new parsing rule. If the new operator is not unary or binary you have to add a function `make_n-ary_formula`. See Section 9.4.4 for further explanations.

For example, the parsing rule for the `BINOP` priority class is:

```
| FORMULA BINOP FORMULA
          { strings = make_binary_formula($2,$1,$3,strings);
            $$=(YYSTYPE)(*strings).entry; }
```

'`| FORMULA BINOP FORMULA`' is the grammar rule for the `BINOP` priority class. The C-function `make_binary_formula` generates the output syntax and stores it in the list `strings`. `$1,$2,$3` refer to the first, the second and the third word of the grammar rule (i.e. `$1` means `FORMULA`). `$$=(YYSTYPE)(*strings).entry;` must appear in the parsing rule of every priority class.

To compose a parsing rule for an n-ary priority class you need the grammar rule and the C-function `make_n-ary_formula`. The grammar rule consists of the token for the priority class, n appearances of `FORMULA` and, if necessary, brackets. The first argument of `make_n-ary_formula` must be the `$`-symbol which refers to the token for the priority class. The last argument must be `strings`. The other arguments are the `$`-symbols which refer to the appearances of `FORMULA`.

9.4.4 Arity

The locations concerning the arity of operators are indicated by the marker `CHANGE_ARITY_HERE`.

As already mentioned in Section 9.4.3 one occurrence is in the file `grammar.y`. Here every parsing rule for an n-ary operator calls a C-function `make_n-ary_formula`.

You also find the `CHANGE_ARITY_HERE` marker in the file `output.c` in the functions for composing formulae from the operators and subformulae. The functions differ in their number of arguments and in the lines

```
str = (char *)malloc(...);
```

and

```
sprintf(...);
```

A function `make_n-ary_formula` looks like this:

```
struct string_list *make_binary_formula(op,fma_1,...,fma_n,strings)
char    *op;
char    *fma_1;
...
char    *fma_n;
struct string_list  *strings;
{
    char *str;
```

```
        str = (char *)malloc(5 + strlen(op) +
        strlen(fma_1) + ... + strlen(fma_n) + 10);
        sprintf(str,"%s(%s,...,%s,l%d)",op,fma_1,...,fma_n,get_label());
        strings = insert_in_string_list(str,strings);
        free(str);

        return(strings);
    }
```

Finally, you have to edit the headerfile `output.h`, find the `CHANGE_ARITY_HERE` marker, and add the line

```
    extern  struct string_list    *make_n-ary_formula();
```

A Commands Reference Manual

This appendix describes briefly all $_3T^AP$ commands. More technical descriptions can be found in Section 5.1. Additional references are given when appropriate in the "Description" parts below.

The syntax of commands is indicated in a form such as:

`prove` [`(`*theorem* [, *KB*]`)`]`.`

Everything in a typewriter font represents those parts of commands that appear in the input exactly as shown. The italicized parts represent parts that vary; the command's description explains their function. Arguments enclosed in square brackets [ ] are optional; they may be omitted, so the command `prove` can also have the form "`prove(`*theorem*`).`" or, when all arguments are omitted, "`prove.`".

For all optional arguments their default values are given.

As usual, parameters that contain characters such as / or .., e.g. path names, have to be enclosed in single quotation marks `' '`.

cd

Syntax	`cd` [`(`*directory*`)`]`.`
Purpose	Changes the working directory.
Argument(s)	*directory*: The new directory (default: the login directory).

compall

Syntax	`compall(`*what_to_prove*`).`
Purpose	Compiles predefined problem sets.
Argument(s)	*what_to_prove*: The problem set(s) to be compiled.
Description	This command proves either one predefined problem set or even all predefined problem sets. The command `compkbx` is used for compiling. The predefined problem sets are listed in Table A.3 (they are described in detail in Chapter 8). Valid instantiations of the argument *what_to_prove* are: • the name of a problem set (e.g. `tests`), • the atom `all` as an abbreviation for the list of all predefined problem sets.
See also	`compkbx`
Example	`compall(all).` Compiles all problem sets.

compkbx

Syntax	`compkbx(`*file*`).`
Purpose	Compiles a knowledge base.
Argument(s)	*file*: The name of the file containing the knowledge base to be compiled.
Description	The compiler is invoked to parse the formulae in *file*. The partial and (if the switch *remove_unlinked* is *on* the complete) information about links is computed (cf. Section 5.12). The compiled knowledge base is written to the file *file*`.kbx`. The command `readkbx` can then be used to load it into the workspace. If *file* does not contain a valid knowledge base (cf. Chapter 3), the compiler fails to parse it and an error message is displayed. Note that it might take *very long* to compute the *complete* information about links for complex knowledge bases.
See also	`readkbx`, `usekbx`
Example	`compkbx('problems/pel23').` Compiles the knowledge base in the file `problems/pel23` and writes the result to the file `problems/pel23.kbx`.

cp

Syntax	`cp(`*oldfile*`,`*newfile*`).`
Purpose	Copies a file.
Argument(s)	*oldfile*: The source file.
	newfile: The target file.
Description	`cp` copies the contents of *oldfile* to *newfile*.
See also	`mv`

date

Syntax	`date.`
Purpose	Displays the date.
See also	`time`

delkb

Syntax	`delkb [(`*KB*`)].`
Purpose	Deletes a knowledge base from the workspace.
Argument(s)	*KB*: The knowledge base to be deleted (default: the current knowledge base).
See also	`delkbs`
Example	`delkb(pel23).` Deletes the knowledge base `pel23` from the workspace.

delkbs

Syntax	`delkbs.`
Purpose	Deletes all knowledge bases from the workspace.
See also	`delkb`

edit

Syntax	`edit(`*file*`).`
Purpose	Calls the editor.
Argument(s)	*file*: The file to be edited.
Description	The editor specified by the parameter *editor* is called to edit the file *file*.
See also	`edopen`

edopen

Syntax	`edit(`*file*`).`
Purpose	Opens a new window, starts the editor in this window.
Argument(s)	*file*: The file to be edited.
Description	First the UNIX command `open` is used to open a new window, then the editor specified by the parameter *editor* is started in this window to edit the file *file*. Note that the `open` command is not available on all systems.
See also	`edit`

get_*varname*

Syntax	`get_`*varname*`(`*variable*`).`
Purpose	Returns the current value of one of $_3T^AP$'s switches and parameters.
Argument(s)	*varname*: The name of the switch of parameter to be read (this is not an argument in the strict sense since it is part of the command's name). *variable*: A Prolog variable that is to be instantiated with the value of the switch or parameter.
Description	These commands can be used to read the value of one of $_3T^AP$'s switches or parameters (Table A); but, in most cases, it might be easier to use the command `lookup`. *variable* has to be an uninstantiated Prolog variable.
See also	`set_`*varname*, `lookup`
Example	`get_maxcounter(MC).` Instantiates the Prolog variable `MC` with the current value of the parameter *maxcounter*.

inconsistent

Syntax	`inconsistent [(`*KB*`)].`

[1] *equality* is by default switched on in the two-valued version and switched off in the many-valued versions.

Switch/Parameter	Type	Range	Default
complete_first	atomic	on, off	off
compute_additional_solutions	atomic	on, off	off
current_kb	alphanumeric	loaded knowledge bases	none
debuglevel	numeric	$0, 1, \ldots, 6$	0
dissbound	numeric	$0, 1, 2, \ldots$	3
disscomplexity	atomic	on, off	off
dissdebuglevel	numeric	$0, 1, \ldots, 6$	0
dissolution	atomic	on, off	off
disspriority	atomic	diss, alpha	diss
editor	alphanumeric	vi, emacs, ...	vi
eqdebuglevel	numeric	$0, 1, \ldots, 6$	0
equality	atomic	on, off	on/off[1]
flattenformulas	atomic	on, off	off
flipconclusion	atomic	on, off	off
grepall	atomic	on, off	on
inc_limit_mbr	numeric	$0, 1, 2, \ldots$	2
inc_limit_mc	numeric	$0, 1, 2, \ldots$	2
kbx_extension	alphanumeric	valid extensions	kbx
maxbranchlength	numeric	$0, 1, 2, \ldots$	20
maxcounter	numeric	$0, 1, 2, \ldots$	2
max_rule_cr_number	numeric	$0, 1, 2, \ldots$	10000
max_rule_simp_number	numeric	$0, 1, 2, \ldots$	10000
max_solutions_per_branch	numeric	$0, 1, 2, \ldots$	10
max_term_number	numeric	$0, 1, 2, \ldots$	10000
outputfile	alphanumeric	valid paths	user
protmode	atomic	on, off	off
stepmode	atomic	on, off	off
tableau_output	atomic	on, off	off
tableauoutfile	alphanumeric	valid paths	tableau.out
tcplus_extension	alphanumeric	valid extensions	kb
removeantilinks	atomic	on, off	off
removeunlinked	atomic	on, off	off
uselemmata	atomic	alpha, on, off	alpha
weight_left_only	atomic	on, off	off

Table A.1: $_3T^AP$'s switches and parameters, their ranges and default values.

Purpose	Tries to prove the inconsistency of a set of axioms.
Argument(s)	*KB*: The knowledge base containing the axiom set (default: the current knowledge base).
Description	The command `inconsistent` has the same effect as the command `prove` if no theorem is specified, i.e., it tries to prove the inconsistency of the axiom set.

See Section 5.2 for a detailed description of the proof procedure. The influence of $_3T^AP$'s various switches and parameters on the proof is summarized in Appendix B.

See also `prove`, `protprove`, `proveinc`, `proveall`

info

Syntax `info` [(*topic*)].

Purpose Provides online information.

Argument(s) *topic*: The topic about which information is to be displayed (default: information about the `info` command).

Description `info` provides information on the topics listed in Table A. The argument *topic* may be chosen from one of these topics; or the atom `all` may be used to get all available information.
`info` uses the current output stream to display the information.

See also `lookup`

Example `info(equality).`
Provides information about the handling of equality.

Topic	Information about
`compiler`	The compiler and the commands for compiling knowledge bases.
`diss`	The dissolution rule.
`equality`	The equality predicate.
`info`	The available information.
`maintain`	Commands for workspace maintenance.
`output`	Tableau output for moreTab and tabTEX.
`prover`	Commands for proving theorems (resp. inconsistency of an axiom set).
`unix`	UNIX-like commands.
`variables`	Switches and parameters.

Table A.2: The topics `info` provides information about.

init

Syntax `init.`

Purpose Initializes the prover.

Description The prover is reset to its initial state, i.e., all switches and parameters are set to their default values (Appendix B), and all knowledge bases are removed from the workspace.

See also `initialize_variables`

initialize_variables

Syntax	`initialize_variables.`
Purpose	Resets all switches and parameters to their default values.
Description	All of $_3T^AP$'s switches and parameters are reset to their default values (cf. Appendix B).
See also	`init`

lookup

Syntax	`lookup.`
Purpose	Displays the current values of $_3T^AP$'s switches and parameters.
Description	This command shows the current values of $_3T^AP$'s switches and parameters (Appendix B). In addition, the names of all loaded knowledge bases are listed. `lookup` sends its output to the current stream.
See also	`info`

ls

Syntax	`ls` [`(`*directory*`)`]`.`
Purpose	Lists the contents of a directory.
Argument(s)	*directory*: The directory to be listed (default: the current working directory).

mv

Syntax	`mv(`*oldpath*`,`*newpath*`).`
Purpose	Moves or renames a file.
Argument(s)	*oldpath*: The old path of the file to be moved. *newpath*: The path the file is to be moved to.
See also	`cp`

protprove

Syntax	`protprove` [`(`*theorem* [, *KB*]`)`]`.`
Purpose	Tries to prove a specified theorem or the inconsistency of a set of axioms; redirects all output to the file specified by *outputfile*.
Argument(s)	*theorem*: The name of the theorem to be proved (default: prove the inconsistency of the axiom set). *KB*: The knowledge base containing *theorem* and the axiom set to be used (default: the current knowledge base).
Description	`protprove` tries to find a prove in exactly the same way as the command `prove`. The difference is that `protprove` writes all output to the file specified by the parameter *outputfile*, whereas `prove` uses the current output stream. Before any output is sent to the file, it is initialized and all information it may contain is lost. Therefore, the old file has to be saved, or another filename must be taken by changing the parameter *outputfile* before `protprove` is called again. To redirect the output of the commands `proveinc` or `inconsistent` one may use the analogue to the sequence of commands

`set_protmode(on), prove(`[*theorem*]`,`[*KB*]`), set_protmode(off).`

of which the command `protprove` is an abbreviation.
See Section 5.2 for a detailed description of the proof procedure. The influence of $_{3}T^{A}P$'s various switches and parameters on the proof is summarized in Appendix B.

See also: `proveinc`, `prove`, `inconsistent`, `proveall`

prove

Syntax: `prove` [(*theorem* [, *KB*])]`.`

Purpose: Tries to prove a specified theorem or the inconsistency of a set of axioms.

Argument(s): *theorem*: The name of the theorem to be proved (default: prove the inconsistency of the axiom set).

KB: The knowledge base containing *theorem* and the axiom set to be used (default: the current knowledge base).

Description: This command starts the proof procedure. If no theorem is specified, $_{3}T^{A}P$ tries to prove the axiom set of the current knowledge base to be inconsistent.
See Section 5.2 for a detailed description of the prove procedure. The influence of $_{3}T^{A}P$'s various switches and parameters on the proof is summarized in Appendix B.

See also: `proveinc`, `protprove`, `inconsistent`, `proveall`

Example: `prove(pel23).`
Tries to prove the theorem `pel23` (using the axiom set of the current knowledge base).

proveall

Syntax: `proveall(`*what_to_prove* [, *parameter*] [, *format*]`).`

Purpose: Proves predefined problem sets; generates a file containing proof statistics.

Argument(s): *what_to_prove*: The problem set(s) to be proved.

parameter: The parameter to be increased (either *maxcounter* or *maxbranchlength*); the abbreviations `mc` and `mbr` can be used (default: *maxcounter*).

format: The format of the statistics file (default: `ascii`).

Description: This command is quite useful for testing $_{3}T^{A}P$. Predefined problem sets are proved automatically; the user neither has to load the knowledge bases, nor set the switches and parameters, nor start the proofs.
The command `proveinc` is used for proving.
The predefined problem sets are listed in Table A.3 (they are described in detail in Chapter 8).
Valid instantiations of the argument *what_to_prove* are:

- the name of a problem set (e.g. `tests`),
- a list of problem sets (e.g. `[pel_prop,pel_pred]`),
- the atom `pelletier` as an abbreviation for `[pel_prop,pel_pred,pel_eq]`,
- the atom `all` as an abbreviation for the list of all predefined problem sets.

The statistical information is written to the file `statistics` in the current directory. Before any output is sent to this file, it is initialized and all information it may contain is lost. Therefore it has to be copied before `proveall` is called again.
The format of the statistics file is denoted by the argument *format*. Two formats are available:

ascii: No special format, easy to read.

tex: A LaTeX-like format that can easily be included into LaTeX documents.

Unless the switch *removeunlinked* is on, the knowledge bases are read using the command `readkbx` (else using `usekbx`). The compiled versions, i.e., the files `*.kbx`, have therefore to be existent (in the directory defined in module `proveall`).
The problem sets and the settings for the switches and parameters are defined in the module `proveall`. See Section 5.1 for a description of how to change these definitions and examples for the usage of `proveall`.

See also `prove`, `proveinc`

Example `proveall(all,mbr,tex).`
Proves all problem sets incrementing *maxbranchlength* and generates a statistics file in LaTeX format.

Name	
`tests`	Simple problems for testing some of $_3T^AP$'s features.
`dagostino`	The problem class given by D'Agostino (D'Agostino, 1990, p. 69).
`mr`	The problem class given by Murray and Rosenthal (Murray & Rosenthal, 1987).
`cr`	The problem class given by Cook and Reckhow (Cook & Reckhow, 1974).
`kalish`	Problems from (Kalish & Montague, 1964).
`meta_pl`	Four problems constructed by means of Morgan's method (Morgan, 1976) of encoding problems from propositional logic in predicate logic.
`pigeon`	The "pigeonhole" problems.
`pig_alt`	An alternate formulation of the "pigeonhole" problems.
`groups`	Two problems from group theory formulated with equality.
`pel_prop`	The problems from propositional logic given by Pelletier (Pelletier, 1986).
`pel_pred`	The problems from predicate logic given by Pelletier (Pelletier, 1986).
`pel_eq`	The problems with equality given by Pelletier (Pelletier, 1986).
`ps`	A class of problems proposed by P. Schmitt.
`phi`	A formula given by Murray & Rosenthal (Murray & Rosenthal, 1993) demonstrating the advantages of dissolution.

Table A.3: The predefined problem sets for the commands `proveall` and `compall`.

proveinc

Syntax	`proveinc(`*theorem* [, *KB*] [, *parameter* [, *init*]]`).`
	`proveinc` [`(`*parameter* [, *init*]`)`]`.`

Purpose	Tries to prove a specified theorem or the inconsistency of a set of axioms by incrementing the specified parameter after each try. The parameter can be either *maxcounter* or *maxbranchlength*.
Argument(s)	*theorem*: The name of the theorem to be proved (default: prove the inconsistency of the axiom set). *KB*: The knowledge base containing *theorem* and the axiom set to be used (default: the current knowledge base). *parameter*: The name of the parameter to be increased (either *maxcounter* or *maxbranchlength*; the abbreviations *mc* and *mbr* can be used (default: *maxcounter*. *init*: The initial value for the specified parameter (default: 0 if the parameter is *maxcounter*, 1 if the parameter is *maxbranchlength*).
Description	$_3T^AP$'s proof procedure is not complete in the strict sense. It might fail to prove a valid theorem if one of the parameters *maxcounter* and *maxbranchlength* is too low. This problem can be overcome by using the command `proveinc`. `proveinc` first sets *maxcounter* (resp. *maxbranchlength*) to the initial value *init* and then iteratively tries to prove the theorem (resp. the inconsistency of the axiom set). After each futile try *maxcounter* or *maxbranchlength* is increased by one. This process goes on until either a proof is found, *maxcounter* or exceeds the value of *inc_limit_mc*, or *maxbranchlength* exceeds *inc_limit_mbr*. `proveinc` should not be used to increment *maxcounter* if all formulae in the knowledge base are solely propositional, since *maxcounter* has no effect on propositional logic proofs.
See also	`prove`, `protprove`, `inconsistent`, `proveall`
Example	`proveinc(pel23,pel23,maxcounter,1).` Tries to prove the theorem `pel23` using the axiom set of the knowledge base `pel23`, starting with *maxcounter* = 1.

pwd

Syntax	`pwd.`
Purpose	Displays the pathname of the current working directory.

readkbx

Syntax	`readkbx(`*file*`).`
Purpose	Loads a knowledge base into the workspace.
Argument(s)	*file*: The name of the file containing the knowledge base to be loaded.
Description	The compiled knowledge base in the file *file*`.kbx` is read into the workspace. The names of the theorems the knowledge base contains are displayed.
See also	`compkbx`, `usekbx`
Example	`readkbx(pel23).` Loads the knowledge base contained in `pel23.kbx` into the workspace.

rm

Syntax	`rm(`*file*`).`
Purpose	Removes a file from the file system.
Argument(s)	*file*: The file to be removed.

set_*varname*

Syntax	`set_`*varname*`(`*value*`).`
Purpose	Assigns a new value to one of $_3T^AP$'s switches or parameters.
Argument(s)	*varname*: The name of the switch or parameter to be set (this is not an argument in the strict sense since it is part of the command's name). *value*: The value to be assigned to the switch or parameter.
Description	The value *value* is assigned to the switch or parameter *varname*. Table A contains a list of all switches and parameters, their ranges and their default values. Consult Appendix B for a detailed description.
See also	`get_`*varname*
Example	`set_maxcounter(2).` Assigns the value 2 to the parameter *maxcounter*.

time

Syntax	`time.`
Purpose	Displays the time.
See also	`date`

usekbx

Syntax	`usekbx(`*file*`).`
Purpose	Compiles a knowledge base and loads it into the workspace.
Argument(s)	*file*: The name of the file containing the knowledge base to be compiled and loaded.
Description	The command `usekbx` is a combination of the commands `compkbx` and `readkbx`, i.e., it first compiles a knowledge base and then loads it into the workspace (for further information consult the descriptions of `compkbx` and `readkbx`).
See also	`compkbx`, `readkbx`
Example	`usekbx(test).` Compiles the knowledge base contained in the file `test`, writes it to the file `test.kbx`, and loads it into the workspace.

writeidx

Syntax	`writeidx` [`(`*KB*`)`]`.`
Purpose	Writes all index entries of a knowledge base to the current output stream.

Argument(s)	*KB*: A knowledge base (default: the current knowledge base).
Description	The index entries, i.e., the partial link information (cf. Section 5.12), of the specified knowledge base is sent to the current output stream.
See also	`writekb`, `writekbx`, `writesort`

writekb

Syntax	`writekb [(`*KB*`)].`
Purpose	Writes all formulae of a knowledge base to the current output stream.
Argument(s)	*KB*: A knowledge base (default: the current knowledge base).
Description	A list of all formulae of the specified knowledge base is sent to the current output stream (in their internal representation, cf. Section 5.5.2).
See also	`writeidx`, `writekbx`, `writesort`

writekbx

Syntax	`writekbx [(`*KB*`)].`
Purpose	Writes all formulae, index entries and sort declarations of a knowledge base to the current output stream.
Argument(s)	*KB*: A knowledge base (default: the current knowledge base).
Description	`writekbx` is a combination of the commands `writekb`, `writeidx`, `writesort`. It sends a list of all formulae, all index entries, i.e., the partial link information, and sort declarations of the specified knowledge base to the current output stream.
See also	`writekb`, `writeidx`, `writesort`

writesort

Syntax	`writesort [(`*KB*`)].`
Purpose	Writes all sort entries of a knowledge base to the current output stream.
Argument(s)	*KB*: A knowledge base (default: the current knowledge base).
Description	A list of all sorts declared in the specified knowledge base (cf. Section 3.2) is sent to the current output stream.
See also	`writekb`, `writekbx`, `writeidx`

B Switches and Parameters

In this appendix all of $_3T^AP$'s switches and parameters are summarized. More technical descriptions can be found in Chapter 5.

compiler_directory

Purpose	The name of the directory the compiler is located in. This parameter is read only.
Type	alphanumeric
Range	valid name
Default	Compiler

complete_first

Purpose	If on, the completion of reduction rules and the normalization of terms are not combined.
Description	If *complete_first* is off, the completion of a reduction system and the normalization of terms are combined (Section 2.6.6.4). In most cases, this is more efficient, then first computing a complete reduction system and then using this completion to compute normal forms. This switch is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches.
Type	atomic
Range	on, off
Default	off
See also	equality

compute_additional_solutions

Purpose	If on, the rules in a complete reduction system are reversed to compute additional solutions to E-unification problems.
Description	If *compute_additional_solutions* is on, and if none of the computed unifiers in the ground-complete set $\mathrm{Sat}(\mathcal{C}(\langle E, s, t\rangle))$ (Theorem 2.55), can be used to close the tableau, the orientation of rules in the completion $\mathcal{R}^\infty$ for E is changed, and the inversion is applied to the unifiers computed so far. This switch is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches.
Type	atomic

Range	on, off
Default	off
See also	equality, max_rule_cr_number, max_rule_simp_number, max_solutions_per_branch, max_term_number

current_kb

Purpose	The knowledge base used as default for all commands.
Description	The current knowledge base is used by default for all commands that have a knowledge base as an optional argument if this argument is omitted (`prove`, `protprove`, `proveinc`, `inconsistent`, `delkb`, `writekb`, `writesort`, `writeidx` and `writekbx`). If *current_kb* is set by `set_current_kb` to a knowledge base not in the workspace, these commands will either fail or have no effect. *current_kb* is set by `readkbx` and `usekbx` to the name of the last knowledge base loaded into the workspace.
Type	alphanumeric
Range	loaded knowledge bases
Default	none

debuglevel

Purpose	Controls the amount of debug information displayed during proofs.
Description	This parameter is for debugging ${}_3T^AP$. If it is set to a value different from 0, debug information is displayed (the higher the value of *debuglevel* the more). The information is sent to the current output stream. In addition, if *stepmode* is on and *debuglevel* is greater than 0, ${}_3T^AP$ will interrupt the proof when a branch has been closed and prompt the user to continue the proof, abort the proof, or change certain parameters (for details see the description of *stepmode*). Debug information is not displayed for parts of a proof that are handled by the modules `dissolution` and `equality`. To get information about dissolution or the closure of branches with the help of equality the parameters *dissdebuglevel* and *eqdebuglevel*, respectively, have to be set to a value different from 0. Since the displayed material is mostly in internal format, the user might prefer using the program moreTab to take a closer look at ${}_3T^AP$'s proofs.
Type	numeric
Range	$0, 1, \ldots, 6$
Default	0
See also	dissdebuglevel, eqdebuglevel, stepmode

dissbound

Purpose	The maximal number of dissolution rule applications on each branch that are based on the same link with respect to their signs.

Description	*dissbound* solves the problem of unfair rule application sequences containing at least one dissolution and one γ-rule application. *dissbound* is a very mighty switch. Its meaning to the module `dissolve` is nearly the same as that of *maxcounter* to the other modules.
Type	numerical
Range	$0, 1, 2, \ldots$
Default	3
See also	disscomplexity, dissolution, disspriority, maxcounter

disscomplexity

Purpose	If on, it is checked which of the two mirror image versions of the dissolution rule yields a less complex new formula.
Description	Since the dissolution rule is asymmetrical, two versions exist that are mirror images of each other. Their application can result in different new formulae with different complexities (cf. Section 5.11.7.1). If *complexity* is on, it is checked which of these two version will probably yield the less complex formula before the dissolution rule is applied (in first-order logic the complexity can only be estimated due to multiple γ-rule applications). Proof times tend to be much longer when this switch is on even if the proofs actually found are short, since the check is quite expensive.
Type	atomic
Range	on, off
Default	off
See also	dissolution, dissbound, disspriority

dissdebuglevel

Purpose	Controls the amount of debug information displayed by `dissolution`.
Description	*dissdebuglevel* provides the possibility to debug the application of the dissolution rule separately. It controls the amount of debug information displayed about the application of the dissolution rule. If *dissdebuglevel* is set to a value different from 0, debug information is displayed (the higher the value of *dissdebuglevel* the more). The information is sent to the current output stream.
Type	numeric
Range	$0, 1, \ldots, 6$
Default	0
See also	debuglevel, eqdebuglevel

dissolution

Purpose	If on, the dissolution rule is available during the proof.

Description	If *dissolution* is switched on, the dissolution rule can be applied to expand or, if possible, close the current branch (cf. Section 5.11.1). As the dissolution rule is only defined for the classical two-valued logic, it can only be used in the two-valued version. In addition, it has only been proved to be complete for propositional logic. Therefore, $_3T^AP$ might fail to prove theorems from predicate logic if *dissolution* is switched on. Since the dissolution rule is not analytical, and links of totally new formulae are not pre-computed, formulae that have been generated by the application of the dissolution rule are, when *removeunlinked* is switched on, though they might have no link, never removed from the branch. In general, if the dissolution rule is applied, much shorter proofs are generated, however, it takes relatively long to test, whether the dissolution rule can be applied due to sformula data structure conversion, so it usually does take longer to find a proof if *dissolution* is switched on.
Type	atomic
Range	on, off
Default	off
See also	disscomplexity, dissbound, disspriority

disspriority

Purpose	Controls the priority of dissolution and α-rule application.
Description	If the switch *disspriority* is set to diss, the dissolution rule is applied, whenever legal. If *disspriority* is set to alpha, α-rules are applied first.
Type	atomic
Range	diss, alpha
Default	diss
See also	disscomplexity, dissolution, dissbound

editor

Purpose	The editor invoked by the `edit` command.
Type	alphanumeric
Range	vi, emacs, ...
Default	vi

eqdebuglevel

Purpose	Controls the amount of debug information displayed by the modules `complete` and `equality`.
Description	*eqdebuglevel* provides the possibility to debug the handling of equality separately. It controls the amount of debug information displayed about the closure of branches using equality. If *eqdebuglevel* is set to a value different from 0, debug information is displayed (the higher the value of *eqdebuglevel* the more). The information is sent to the current output stream.

Type	numeric
Range	$0, 1, \ldots, 6$
Default	0
See also	debuglevel, dissdebuglevel

equality

Purpose	If on, equality is used to close branches.
Description	If *equality* is switched on, the `complete` module is invoked if a branch is exhausted and cannot be closed without equality applications. In that case the equality predicate `=` is interpreted as the identity relation. Therefore, if a proof is found, one can conclude that the proved theorem is valid in all normal models (not necessarily in all models). Demodulators, i.e., formulae of the form $t == s$, that are present on a branch, are applied from left to right to all terms they can be applied to. It is left to the user to define demodulators in such a way that completeness is preserved. If *equality* is switched off, the predicates `=` and `==` are treated as ordinary binary predicates. The switch *equality* influences the handling of links (cf. the description of the parameter *removeunlinked*). Since the semantics of a multiple-valued equality predicate is not clear, *equality* is by default switched off in the many-valued versions. The modules `complete` and `equality` ignore all sort information. Therefore, *equality* should be switches off if more than one sort is defined; otherwise, incorrect proofs may be generated.
Type	atomic
Range	on, off
Default	on (two-valued version), off (many-valued versions)

flattenformulas

Purpose	Controls pre-processing of formulae.
Description	Only experienced users should switch *flattenformulas* on; see Section 5.12.5.
Type	atomic
Range	on, off
Default	off
See also	removeantilinks

flipconclusion

Purpose	If on, the processing order of newly generated subbranches is reversed.
Description	If *flipconclusion* is switched off, branches are closed in the order that is defined by the tableau rules in the module `rules`, i.e., the branches of the tableau built using these rules are closed from left to right. If *flipconclusion* is switched on, the branches are closed in to reverse order, i.e., from right to left.

	flipconclusion can have an influence on the tableau built as well as on the time needed to find a proof. The reason is that backtracking can be avoided if those branches that are more difficult to close are closed first.
Type	atomic
Range	on, off
Default	off

grepall

Purpose	If on, all formulae that are linked to the negation of the theorem to be proved are put on the initial tableau.
Description	If *grepall* is on and a theorem is to be proved, not only the negated theorem is put on the initial tableau but in addition all formulae in the knowledge base that are linked to the negated theorem.
Type	atomic
Range	on, off
Default	on

inc_limit_mbr

Purpose	The maximal value that is used by the command `proveinc` for the parameter *maxbranchlength*.
Description	The command `proveinc` iteratively tries to find a proof and increases either *maxcounter* or *maxbranchlength* by one after each futile try until either a proof is found or *maxbranchlength* exceeds the limit *inc_limit_mbr* (or *maxcounter* exceeds *inc_limit_mc*). See Appendix A for a description of the command `proveinc`.
Type	numeric
Range	$0, 1, \ldots$
Default	20
See also	maxbranchlength, inc_limit_mc

inc_limit_mc

Purpose	The maximal value that is used by the command `proveinc` for the parameter *maxcounter*.
Description	The command `proveinc` iteratively tries to find a proof and increases either *maxcounter* or *maxbranchlength* by one after each futile try until either a proof is found or *maxcounter* exceeds the limit *inc_limit_mc* (or *maxbranchlength* exceeds *inc_limit_mbr*). See Appendix A for a description of the command `proveinc`.
Type	numeric
Range	$0, 1, \ldots$
Default	2
See also	maxcounter, inc_limit_mbr

kbx_extension

Purpose	The file name extension used for compiled knowledge bases.
Description	This extension is used by the commands `compkbx` and `usekbx` for files containing compiled knowledge bases.
Type	alphanumeric
Range	valid extensions
Default	kbx

max_rule_cr_number

Purpose	The maximal number of applications of the critical pair rule per branch.
Description	This parameter is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches (see Section 5.13.6).
Type	numeric
Range	$0, 1, \ldots$
Default	10000
See also	equality, max_rule_simp_number, max_solutions_per_branch, max_term_number

max_rule_simp_number

Purpose	The maximal number of applications of the composition and the simplification rule per branch.
Description	This parameter is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches (see Section 5.13.6).
Type	numeric
Range	$0, 1, \ldots$
Default	10000
See also	equality, max_rule_cr_number, max_solutions_per_branch, max_term_number

max_solutions_per_branch

Purpose	The maximal number of closing substitutions that are computed for a branch using equality.
Description	This switch is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches (see Section 5.13.6).
Type	numeric
Range	$0, 1, \ldots$
Default	10
See also	equality, max_rule_cr_number, max_rule_simp_number, max_term_number

max_term_number

Purpose	The maximal number of new constrained terms that are derived during the computation for closing a single branch.
Description	This switch is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches (see Section 5.13.6).
Type	numeric
Range	$0, 1, \ldots$
Default	10000
See also	equality, max_rule_cr_number, max_rule_simp_number, max_solutions_per_branch

maxbranchlength

Purpose	*maxbranchlength* restricts the maximal length of the branches. If *maxbranchlength* is set to 0, there is no restriction.
Description	*maxbranchlength* restricts the length of the branches. Branches that are longer than *maxbranchlength* are not further expanded. *maxbranchlength* $= 0$ means that the restriction is switched off. If the length of branches is restricted, $_3T^AP$'s proof procedure is not complete in the strict sense. It might fail to prove a valid theorem if the parameter *maxbranchlength* is too low. To overcome this problem, the command `proveinc` may be used. The length of branches might slightly exceed *maxbranchlength*. In fact, the maximal length of branches is $$\mathit{maxbranchlength} + \mathit{maximal\ length\ of\ extensions} - 1\ .$$[1]
Type	numeric
Range	$0, 1, \ldots$
Default	0
See also	maxcounter, inc_limit_mbr

maxcounter

Purpose	The maximal number of γ-rule applications to a γ-formula minus one.
Description	*maxcounter* is the maximal number minus one of γ-rule applications to a γ-formula on each branch it occurs on, i.e., if *maxcounter* is set to 0, the corresponding γ-rule is applied exactly once to each γ-formula on each branch. Because of this restriction $_3T^AP$'s prove procedure is not complete in the strict sense. It might fail to prove a valid theorem if the parameter *maxcounter* is too low. To overcome this problem the command `proveinc` may be used (cf. Appendix A). Keep in mind that *maxcounter* is probably $_3T^AP$'s most important parameter. In most cases, the length of predicate logic proofs as well as the time $_3T^AP$ needs to find them decreases drastically if *maxcounter* is set to a smaller value (provided a proof exists for that smaller value); in particular, if backtracking occurs or

[1] That is *maxbranchlength* $+ 1$ in two-valued logic.

	equality is involved, there is usually a considerable effect on the time needed to find a proof. One should, therefore, try smaller values for *maxcounter* first (this can be done automatically by using the command `proveinc`). *maxcounter* has no effect on propositional logic proofs.
Type	numeric
Range	$0, 1, \ldots$
Default	2
See also	maxbranchlength, inc_limit_mc

outputfile

Purpose	The name of the file $_3T^AP$'s output is redirected to if *protmode* is on.
Description	If *protmode* is switched on all output that is usually sent to the current output stream is redirected to the file specified by *outputfile*. The file is initialized and any information it may contain is lost when *protmode* is switched on; it is closed by switching *protmode* off. This file is also used by the command `protprove`, that automatically switches *protmode* on and off. Note that *outputfile* is different from *tableauoutfile* (the proof protocol that can be used as an input for the utility programs moreTab and tabTEX).
Type	alphanumeric
Range	valid paths
Default	user
See also	protmode, tableau_output, tableauoutfile

protmode

Purpose	If on, $_3T^AP$'s output is redirected to the file specified by *outputfile*.
Description	If *protmode* is switched on all output that is usually sent to the current output stream is redirected to the file specified by *outputfile*. The file is initialized and any information it may contain is lost when *protmode* is switched on; it is closed by switching *protmode* off. *protmode* is automatically switched on and off by the command `protprove`. Note that *protmode* is different from *tableau_output* (a proof protocol that can be used as an input for the utility programs moreTab and tabTEX is generated if *tableau_output* is on).
Type	atomic
Range	on, off
Default	off
See also	outputfile, tableau_output, tableauoutfile

removeantilinks

Purpose	Controls pre-processing of formulae.

Description	Only experienced users should switch *removeantilinks* on; see Section 5.12.5.
Type	atomic
Range	on, off
Default	off
See also	flattenformulas

removeunlinked

Purpose: If on, formulae on a branch that do not have a link are removed.

Description: If the switch *removeunlinked* is on, formulae F are removed that are not potentially involved in the closure of the current branch, i.e., that do neither

- have a link to a formula on the current branch (including the formula F itself), nor
- have a link to a formula in the knowledge base that, up to that point, has not been put on the branch, nor
- contain the equality predicate or the demodulator predicate (this last point is only taken into concern if the switch *equality* is on).

Formulae are actually not checked for links and, if possible, removed when they are put on a branch, but, what is more effective, when they have been chosen for rule application (cf. Section 5.4).

The information about links is pre-compiled, i.e., if (and only if) *removeunlinked* is on, a complete list of all existing links is generated when the knowledge base is compiled by `compkbx` or `usekbx`. This list is included in the generated `.kbx` file (cf. Section 5.12).

Since it may take very long to generate the complete list of links for more complex knowledge bases, this is only done if *removeunlinked* is on. Therefore, a knowledge base that has been compiled with *removeunlinked* switched off has to be recompiled using `usekbx` before a proof with *removeunlinked* switched on is started.

In general switching *removeunlinked* on shortens proofs but, since it is quite expensive to check formulae for links, it then usually takes longer to find a proof.

Since the dissolution rule is not analytical, and links of totally new formulae are not pre-computed, formulae that have been generated by the application of the dissolution rule are, when *removeunlinked* is switched on, though they might have no link, never removed from the branch.

See Section 5.12.3.1 for a discussion of the theoretical aspects of using links.

Type	atomic
Range	on, off
Default	off

stepmode

Purpose: If on and *debuglevel* is greater than 0, proofs are interrupted at certain points.

Description: If *stepmode* is on and *debuglevel* is greater than 0, $_3T^AP$ will interrupt a proof when a branch has been closed, and, if *debuglevel* is greater than 2, when a formula has been chosen for rule application.

The user is then prompted for an input. The following commands are available:

`c`	Continue	Continue the proof.
`a`	Abort	Abort the proof.
`h`	Help	List the available commands.
`l`	Leap	Switch *stepmode* off.
`o`	Dissolution off	Switch *dissolution* off.
`d`	Debugging off	Set *debuglevel* to 0.
`e`	Equality debugging off	Set *eqdebuglevel* to 0.
`n`	Dissolution debugging off	Set *dissdebuglevel* to 0.

Type	atomic
Range	on, off
Default	off
See also	debuglevel

tableauoutfile

Purpose	The name of the file a proof protocol is written to if *tableau_output* is on.
Description	If *tableauoutfile* is switched on, a proof protocol is written to the file specified by *tableauoutfile*. This file can be used as an input file for the utility programs moreTab and tabTEX (cf. Chapter 6). The file is initialized (and all information the file may contain is lost), when *tableau_output* is switched on and every time a new proof is started. The file is closed when a proof is found. If the search for a proof has been interrupted by the user, the proof protocol might be incorrect. Note that *tableauoutfile* is different from *outputfile* (the output usually sent to the current stream is redirected to the file specified by *outputfile* if *protmode* is on).
Type	alphanumeric
Range	valid paths
Default	tableau.out
See also	tableau_output, outputfile, protmode

tableau_output

Purpose	If on, a proof protocol is written to the file specified by *tableauoutfile* that can be used as an input for moreTab and tabTEX.
Description	If *tableau_output* is switched on, a proof protocol is written to the file specified by *tableauoutfile*. This protocol file can be used as an input file for the utility programs moreTab and tabTEX (cf. Chapter 6). The file is initialized (and all information the file may contain is lost), when *tableau_output* is switched on and every time a new proof is started. The file is closed when a proof is found. If the search for a proof has been interrupted by the user, the proof protocol might be incorrect. Note that *tableau_output* is different from *protmode* (the output usually sent to the current stream is redirected to the file specified by *outputfile* if *protmode* is on).
Type	atomic
Range	on, off

Default	off
See also	tableauoutfile, protmode, outputfile

tcplus_extension

Purpose	The file name extension used by the compiler for temporary files.
Type	alphanumeric
Range	valid extensions
Default	kb

uselemmata

Purpose — Controls the generation of lemmata.

Description — If an extension B' of a branch B has been closed and is thus not satisfiable, one can take advantage of that knowledge by generating lemmata and adding those lemmata to other extensions B'' of B. For a discussion of the theoretical background of lemma generation see Section 2.4.
The lemmata to be added to a branch are defined by the predicate `get_lemmata` in the module `rules` (cf. Section 5.10).
The three possible values alpha, on and off have the following meanings:

alpha: Only those lemmata are added to a branch that do not immediately result in several subbranches if a tableau rule is applied to them, e.g. in the two-valued version lemmata are added that are not a β-formula.

on: All lemmata defined by `get_lemmata` are added.

off: No lemmata are added.

In most cases the value alpha brings the best results. To set *uselemmata* to on or to off will only rarely lead to shorter proofs. Despite of that, some problems from predicate logic can only be proved with *uselemmata* switched off (e.g. Pelletier's 46th problem, cf. Section 8.1.10).
Also in some of the currently available many-valued versions of the `rules` module lemma generating information is present.

Type	atomic
Range	alpha, on, off
Default	alpha

weight_left_only

Purpose — If on, the weight of a constrained rule does not include the weight of its right side.

Description — The weight of a constrainted term is the number of function symbols, constant symbols, variables, and logical operators occurring in it (including its constraint). If *weight_left_only* is on, only the symbols in the left side and in the constraint of a rule are counted. See Section 5.13.5 on how to change the definition of the term weight.
This switch is meaningful only if the switch *equality* is on, i.e. E-unification problems are solved to close branches.

Type	atomic
Range	on, off
Default	off
See also	equality

C Installation

C.1 $_3T^AP$'s Main Parts

$_3T^AP$ consists of the following main parts that have to be installed separately (there is a `makefile` that allows to install alls parts automatically, see Section C.3):

- The prover itself (Section C.4),
- the compiler (Section C.5),
- utilities for visualizing proofs (Section C.6),
- predefined problem sets.

This chapter supplies a guide for installing the $_3T^AP$ system as a stand-alone system[1] in a Quintus Prolog or SICStus Prolog environment on a supported machine and operating system[2] and gives some hints to port $_3T^AP$ to other machines and/or different Prolog environments.

C.2 $_3T^AP$'s Various Files

No special directory structure is assumed by the $_3T^AP$ system. The only constraint is that all the $_3T^AP$ sources are placed in the same directory and that all the compiler sources are in one directory, too. They need not be located in the same directory. But if you want work with $_3T^AP$ on various logics it is better to place these parts of the system in different locations since the compiler may be used for all your logics and thus some disk space is saved.

Table C.1 shows the default directory structure for a $_3T^AP$ system that is used for classical two-valued logic (subdirectory `2version`), for three-valued logics (subdirectory `3version`), and for a seven-valued logic (subdirectory `7version`). The `problems` subdirectories may be used to hold the problems in $_3T^AP$'s input syntax.

Table C.2 lists the necessary files which would have been placed in `threetap/2version` and with some modifications in `declarations` and `rules` concerning the logic in `threetap/3version` and `threetap/7version`.[3] The files in the compiler directory are shown in Table C.3. The sources files for the utilities moreTab and tabTEX are listed in Tables C.4 and C.5.

The location of the compiler has to be made known to the prover; there are two possibilities for this (no modifications are necessary if the default directory structure is used):

[1] If $_3T^AP$ is to be installed as a part of LILOG–KR, the only change to be made is that `boot_tcg` has to be used for compiling $_3T^AP$ from Prolog, instead of `boot`. Please refer to the LILOG–KR manuals for the proper place of calling `boot_tcg`.

[2] These are currently: Quintus Prolog 3.0, Quintus Prolog 3.1 and SICStus Prolog 2.1 on SUN Sparc under SunOS 4.1.x.

[3] In addition, the directory `2version` contains the files `eqinterf.pl` and `comset.pl`, which are necessary for using the equality handling method stand-alone (see Section C.7).

$$\begin{tikzcd}
 & & \mathit{threetap} \arrow[r] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[ddr] & \mathit{Compiler} \\
2\mathit{version} \arrow[d] & 3\mathit{version} \arrow[d] & 7\mathit{version} \arrow[d] & \mathit{Moretab} \\
\mathit{problems} & \mathit{problems} & \mathit{problems} & \mathit{Tabtex}
\end{tikzcd}$$

Table C.1: The default directory structure of the ${}_3T^AP$ system.

boot.pl	boot_tcg.pl	choice.pl
closure.pl	complete.pl	datastructures.pl
declarations.pl	dissolve.pl	equality.pl
globalvars.c	globalvars_quintus.c	globalvars_sicstus.c
heuristics.pl	index.pl	inference.pl
information.pl	interface.pl	main.pl
makekbx.pl	msg_tap.pl	output.pl
preproc.pl	proveall.pl	rules.pl
sysdep.pl	unification.pl	

Table C.2: ${}_3T^AP$ sources (contents of `2version`, `3version` etc.).

grammar.y	output.c	output.h
scanner.l		

Table C.3: Source files in the compiler directory.

`moretab.c`	`moretab.x`	`token.h`

Table C.4: moreTab sources.

`tabtex.c`	`tabtex.h`	`tabtex.x`

Table C.5: tabTEX sources.

- Setting the variable `compdir` in the module `globalvars.c` accordingly (its pre-defined value is `../Compiler`; it is, however, better to use an absolute filename).
- Making the compiler directory part of the Unix shell's search path.

C.3 Installing ${}_3T^AP$ Using the Makefile

A `makefile` is provided for compiling and installing all parts of ${}_3T^AP$ automatically. The `makefile` should be (and usually is) placed in the parent directory `threetap` of the directories `2version`, `3version`, `Compiler`, etc. (see Table C.1).

You probably will have to edit the `makefile` before using it, in particular, if you are not using SICStus Prolog (which is the default) but Quintus Prolog. The following macros have to be defined according to the installation of your local Prolog system:

`PROLOG_TYPE` has to be set to either `QUINTUS` or `SICSTUS`.

`PROLOG_CMD` has to be set to the command for calling the Prolog compiler (usually `sicstus` for SICStus and `prolog` for Quintus Prolog).

`PROLOG_NAME` has to be set to either `quintus` or `sicstus`.

`HEADER_FILE_PATH` has to be set to the name of the path containing the headerfile `quintus.h` (resp. `sicstus.h`). These header files are part of the Quintus Prolog and SICStus Prolog distributions, respectively.

If you are not using the default directory structure (Table C.1), you have to change the definitions of the macros `TWODIR`, `THREEDIR`, `SEVENDIR`, `COMPDIR`, `MORETABDIR`, and `TABTEXDIR` as well.

Once the macros are defined appropriately, the complete ${}_3T^AP$ system can be installed by just typing the shell command:

```
[~] > make
```

Besides that, you can use the commands

`make 2version` to install the two-valued version separately.

`make 3version` to install the three-valued version separately.

`make 7version` to install the seven-valued version separately.

`make compiler` to install the compiler separately.

`make moretab` to install the moreTab utility separately.

`make tabtex` to install the tabTEX utility separately.

`make clean` to remove all temporary files not needed for running ${}_3T^AP$.

`make xclean` to remove all files (including the executables) generated by the makefile.

C.4 Installing the Prover

First, the C-object files `globalvars_quintus.o` and `globalvars_sicstus.o` have to be generated. They are included by `sysdep.pl` when initializing the C foreign language interface. With most C compilers this may be achieved by switching to the directory `2version` (resp. `3version` or `7version`) and typing

```
[~/2version] > cc -c -Iquintus_header_file_path globalvars_quintus.c
```

resp.

```
[~/2version] > cc -c -Isicstus_header_file_path globalvars_sicstus.c
```

`cc` is the name of the default C compiler on most machines. The `-c` switch prevents a call to the system linker, thus only the object files `globalvars_quintus.o` resp. `globalvars_sicstus.o` are generated. *quintus_header_file_path* (resp. *sicstus_header_file_path*) is the path containing the headerfile `quintus.h` (resp. `sicstus.h`). These header files are part of the Quintus Prolog and SICStus Prolog distributions, respectively.

Now, the prover may be compiled by consulting (or compiling) `boot.pl` after the Prolog system has been started. For example:

```
[~/2version] > prolog

Quintus Prolog Release 3.0 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California U.S.A. (415) 965-7700

| ?- compile(boot).
```

After a few seconds (or a few minutes on machines with lower performance) $_3T^AP$ will be ready:

```
Available Information Pages:

Compiler:              see info(compiler)
Prover:                see info(prover)
Equality:              see info(equality)
Dissolution:           see info(diss)
Maintain Workspace:    see info(maintain)
Variables:             see info(variables)
Tableau output:        see info(output)
Unix:                  see info(unix)
Info:                  see info for this help

All:                   info(all) prints all available info pages

% boot.pl compiled in module user, 78.850 sec 530,812 bytes

yes
| ?-
```

Possibly you got some warnings saying that the clauses for the predicates `lookup`/n (where $n = 2, \ldots, 6$) are not together in the source file, and that the predicates `cd` and `ls` have already been imported into the module `sysdep`. These warnings can safely be ignored.

Quintus Prolog users may now produce an executable image of $_3T^AP$ by the call

```
save_program(tap).
```

and SICStus users by the call

```
save(tap).
```

After that you can start the prover from your favourite shell by calling `tap` from the appropriate directory (`2version` in the above example) without compiling `boot.pl` or anything else.

C.5 Installing the Compiler

The compiler, i.e. the executable file `parser`, is generated by using the Unix tools Lex and Yacc (or by using Flex and Bison[4]).

Switch to the directory `Compiler`, and use the shell commands

```
[~/Compiler] > lex scanner.l
[~/Compiler] > yacc -d grammar.y
[~/Compiler] > cc y.tab.c lex.yy.c output.c -ll -o parser
```

to compile `scanner.l` using Lex, to compile `grammar.y` using Yacc, and finally to generate `parser` using the C-compiler.

To generate the executable file `debug` (see Section 5.15.3) use the shell command

```
[~/Compiler] > cc -D DEBUG lex.yy.c -ll -o debug
```

(after compiling `scanner.l`).

Note, that the name of the directory containing the executable `parser` has to be made known to the prover (see Section C.2).

C.6 Installing the Utilities for Visualizing Proofs

You need the Unix tool Flex to compile the tools for visualizing proofs: moreTab is compiled by changing to the directory containing the moreTab files (Table C.4) and typing:

```
[~/Moretab] > flex moretab.x
[~/Moretab] > cc moretab.c lex.yy.c -ll -o moretab
```

A file `moretab` will appear in the directory which is the executable moreTab.

tabTEX is (analogously) compiled by changing to the directory containing the tabTEX files (Table C.5) and typing

```
[~/Tabtex] > flex tabtex.x
[~/Tabtex] > cc tabtex.c lex.yy.c -ll -o tabtex
```

A file `tabtex` will appear in the directory which is the executable tabTEX.

[4] You can use "`flex -l`" instead of "`lex`" and "`bison -y`" instead of "`yacc`".

C.7 Using the Equality Handling Method Stand-alone

It is possible to use $_3T^AP$'s completion-based method for solving mixed and rigid E-unification problems stand-alone. It can, thus, be used to add equality to other implementations of Gentzen-type calculi.

For that purpose `eqinterf.pl` provides an interface to module `complete`. It implements the predicate

```
close_conjunctive_path(+Equalities,+Pos_literals,+Neg_literals,
                       +Inequalities,+Univ_vars)
```

This predicate searches for solutions to the simultaneous mixed E-unification problems, that would be extracted from a branch containing:[5]

- the equalities in the list `Equalities`, where a single equality is represented by `[s=t,Univ]`; `Univ` is a list of the variables with respect to which the equality is universal;
- the (positive) literals in the list `Pos_literals` (in addition to the equalities);
- the negation of the literals in the list `Neg_literals` (in addition to the inequalities);
- the negation of the equalities in the list `Inequalities`.

Positive and negative literals and inequalities are simple Prolog terms (without signing and without sorts). `Univ_vars` is the list of all universal variables occurring in the problem.

If a solution is found, `close_conjunctive_path` applies the necessary instantiations and succeeds; backtracking is possible.

The values of the $_3T^AP$ parameters that influence the handling of equality, namely *complete_first*, *compute_additional_solutions*, *weight_left_only*, *max_rule_cr_number*, *max_rule_simp_number*, *solutions_per_branch*, and *max_term_number*, are defined in `comset.pl` if the equality handling method is used stand-alone.

[5] Which E-unification problems are extracted is described in Section 2.6.5, the method for solving these problems in Section 2.6.6, and its implementation in Section 5.13.

References

BACHMAIR, LEO, DERSHOWITZ, NACHUM, & PLAISTED, DAVID A. 1989. Completion without Failure. *Chap. 1 of:* AÏT-KACI, H., & NIVAT, M. (eds), *Resolution of Equations in Algebraic Structures, Volume 2.* Academic Press.

BECKERT, BERNHARD. 1991 (July). *Konzeption und Implementierung von Gleichheit für einen tableau-basierten Theorembeweiser.* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

BECKERT, BERNHARD. 1992 (Jan.). *Konzeption und Implementierung von Gleichheit für einen tableau-basierten Theorembeweiser.* IWBS Report 208. IBM Germany, Institute for Knowledge Based Systems.

BECKERT, BERNHARD. 1993a (Mar.). A Completion-Based Method for Adding Equality to Free Variable Semantic Tableaux. *Pages 19–22 of:* BASIN, D., HÄHNLE, R., FRONHÖFER, B., POSEGGA, J., & SCHWIND, C. (eds), *Proceedings, 2nd Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Marseille.*

BECKERT, BERNHARD. 1993b (July). *Ein vervollständigungsbasiertes Verfahren zur Behandlung von Gleichheit im Tableaukalkül mit freien Variablen.* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

BECKERT, BERNHARD. 1994a. Adding Equality to Semantic Tableaux. *Pages 29–42 of:* BRODA, K., D'AGOSTINO, M., GORÉ, R., JOHNSON, R., & REEVES, S. (eds), *Proceedings, 3rd Workshop on Theorem Proving with Analytic Tableaux and Related Methods.* Imperial College London, Department of Computing, Tech Report TR-94/5.

BECKERT, BERNHARD. 1994b. A completion-based method for mixed universal and rigid E-unification. *Pages 678–692 of:* BUNDY, ALAN (ed), *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy/France.* LNAI 814. Springer Verlag.

BECKERT, BERNHARD. 1994c. Using E-Unification to Handle Equality in Universal Formula Semantic Tableaux. *In:* BAUMGARTNER, P., BÜRCKERT, H.-J., COMON, H., FRISCH, A., FURBACH, U., MURRAY, N., PETERMANN, U., & STICKEL, M. (eds), *Proceedings, Theory Reasoning in Automated Deduction, Workshop at CADE-12, Nancy, France.*

BECKERT, BERNHARD, & HÄHNLE, REINER. 1992. An Improved Method for Adding Equality to Free Variable Semantic Tableau. *Pages 507–521 of:* KAPUR, D. (ed), *Proceedings, 11th Conference on Automated Deduction CADE, Albany/NY.* Springer, LNCS 607.

BECKERT, BERNHARD, & POSEGGA, JOACHIM. 1994a (Apr.). Lean Theorem Proving: Maximal Efficiency from Minimal Means (Position Paper). *Pages 7–8 of: Working Notes, AISB Workshop "Automated Reasoning: Closing the Gap between Theory and Practice", Leeds, England.*

BECKERT, BERNHARD, & POSEGGA, JOACHIM. 1994b. leanT^AP: Lean Tableau-based Deduction. *Journal of Automated Reasoning.* To appear.

Beckert, Bernhard, & Posegga, Joachim. 1994c. leanT^AP: Lean Tableau-Based Theorem Proving. Extended Abstract. *Pages 793–797 of:* Bundy, A. (ed), *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France.* Springer, LNCS 814.

Beckert, Bernhard, Hähnle, Reiner, & Schmitt, Peter H. 1993. The Even More Liberalized δ-Rule in Free Variable Semantic Tableaux. *Pages 108–119 of:* Gottlob, G., Leitsch, A., & Mundici, D. (eds), *Proceedings, 3rd Kurt Gödel Colloquium (KGC), Brno, Czech Republic.* Springer, LNCS 713.

Beckert, Bernhard, Hähnle, Reiner, Ramesh, Anavai, & Murray, Neil V. 1994. On Anti-Links. *Pages 275–289 of:* Pfenning, F. (ed), *Proceedings, 5th International Conference on Logic Programming and Automated Reasoning (LPAR), Kiev, Ukraine.* Springer, LNCS 822.

Beth, E. W. 1986. Semantic Entailment and Formal Derivability. *Pages 262–266 of:* Berka, Karel, & Kreiser, Lothar (eds), *Logik-Texte. Kommentierte Auswahl zur Geschichte der modernen Logik.* Berlin: Akademie-Verlag.

Bibel, Wolfgang. 1987. *Automated Theorem Proving.* Second revised edn. Vieweg, Braunschweig.

Carnielli, Walter A. 1987. Systematization of Finite Many-Valued Logics through the Method of Tableaux. *Journal of Symbolic Logic*, **52**(2), 473–493.

Chabin, Jacques, Anantharaman, Siva, & Réty, Pierre. 1993 (July). *E–Unification via Constraint Rewriting.* Unpublished.

Comon, Hubert. 1990. Solving Inequations in Term Algebras. *In: Proceedings,5th Annual IEEE Symposium on Logic in Computer Science (LICS), Philadelphia, PA.* IEEE Computer Society Press.

Cook, Stephen, & Reckhow, Robert. 1974. On the lengths of proofs in the propositional calculus. *Pages 135–148 of: Proceedings, 6th STOC.*

D'Agostino, Marcello. 1990 (Nov.). *Investigations into the Complexity of some Propositional Calculi.* Ph.D. thesis, Oxford University Computing Laboratory, Programming Research Group. Also Technical Monograph PRG–88, Oxford University Computing Laboratory.

Dershowitz, Nachum. 1987. Termination of Rewriting. *Journal of Symbolic Computation*, **3**(1), 69–115.

Doherty, Patrick. 1991. A constraint-based approach to proof procedures for multi-valued logics. *In: First World Conference on the Fundamentals of Artificial Intelligence WOCFAI–91, Paris.*

Fenstad, Jens Erik, Halvorsen, Per-Kristian, Langholm, Tore, & van Benthem, Johan F. A. K. 1985. *Equations, Schemata and Situations: A Framework for Linguistic Semantics.* Tech. rept. CSLI–85–29. Center for the Studies of Language and Information Stanford.

Fitting, Melvin C. 1990. *First-Order Logic and Automated Theorem Proving.* Springer, New York.

Gallier, Jean H., Narendran, Paliath, Raatz, Stan, & Snyder, Wayne. 1992. Theorem Proving Using Equational Matings and Rigid E–Unification. *Journal of the ACM*, **39**(2), 377–429.

GERBERDING, STEFAN. 1990 (July). *Exploration der Quintus-Prolog-C-Schnittstelle und Entwicklung von C-Werkzeugen für einen automatischen Beweiser.* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

GERBERDING, STEFAN. 1991 (Dec.). *Monomorphe Axiomatisierung von Intervallarithmetiken mit mehrwertigen Logiken.* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

GOUBAULT, J. 1993. *Simultaneous Rigid E-Unifiability is NEXPTIME-Complete.* Technical Report. Bull Corporate Research Center.

HÄHNLE, REINER. 1990a. *Spezifikation eines Theorembeweisers für dreiwertige First–Order Logik.* IWBS Report 136. Wissenschaftliches Zentrum, IWBS, IBM Deutschland.

HÄHNLE, REINER. 1990b. Towards an Efficient Tableau Proof Procedure for Multiple-Valued Logics. *Pages 248–260 of: Proceedings, Workshop on Computer Science Logic (CSL), Heidelberg.* Springer, LNCS 533.

HÄHNLE, REINER. 1991. Uniform Notation of Tableaux Rules for Multiple-Valued Logics. *Pages 238–245 of: Proceedings, International Symposium on Multiple-Valued Logic (ISMVL), Victoria.* IEEE Press.

HÄHNLE, REINER. 1992a. Analytic Tableaux and Integer Programming. *In:* FRONHÖFER, B., HÄHNLE, R., & KÄUFL, TH. (eds), *Proceedings, Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Lautenbach/Germany.* Internal Report 8/92, University of Karlsruhe.

HÄHNLE, REINER. 1992b. A New Translation from Deduction into Integer Programming. *In: Proceedings, Conference on Artificial Intelligence and Symbolic Mathematical Computations, Karlsruhe.* Springer, LNCS.

HÄHNLE, REINER. 1992c (May). *Tableaux-Based Theorem Proving in Multiple-Valued Logics.* Ph.D. thesis, University of Karlsruhe, Dept. of Computer Science.

HÄHNLE, REINER. 1993a. *Automated Deduction in Multiple-Valued Logics.* International Series of Monographs on Computer Science, vol. 10. Oxford University Press.

HÄHNLE, REINER. 1993b. Efficient Deduction in Many-Valued Logics. *Pages 54–61 of:* JACKSON, PETER, & SCHERL, RICHARD (eds), *Proceedings, Workshop Automated Deduction in Nonstandard Logics, AAAI Fall Symposium Series, Raleigh/NC, USA.* AAAI Technical Report, nos. FS–93–01. AAAI.

HÄHNLE, REINER. 1993c. Short Normal Forms for Arbitrary Finitely-Valued Logics. *In: Proceedings, ISMIS'93, Trondheim, Norway.* Springer, LNCS.

HÄHNLE, REINER. 1994a. Efficient Deduction in Many-Valued Logics. *Pages 240–249 of: Proceedings, International Symposium on Multiple-Valued Logics (ISMVL'94), Boston/MA, USA.* IEEE Press, Los Alamitos.

HÄHNLE, REINER. 1994b. Short Conjunctive Normal Forms in Finitely-Valued Logics. *Journal of Logic and Computation.* To appear.

HÄHNLE, REINER, & KERNIG, WERNER. 1993. Verification of Switch Level Designs with Many-Valued Logic. *Pages 158–169 of:* VORONKOV, A. (ed), *Proceedings, 4th International Conference on Logic Programming and Automated Reasoning (LPAR'93), St. Petersburg.* Springer, LNAI 698.

HÄHNLE, REINER, & SCHMITT, PETER H. 1993. The liberalized δ-rule in free variable semantic tableaux. *Journal of Automated Reasoning.* To appear.

HINTIKKA, K. J. J. 1955. Form and Content in Quantification Theory. *Acta Philosohica Fennica*, **8**, 7–55.

JEFFREY, RICHARD C. 1967. *Formal Logic. Its Scope and Limits.* McGraw-Hill, New York.

KALISH, DONALD, & MONTAGUE, RICHARD. 1964. *Logic Techniques of Formal Reasoning.* Harcourt, Brace & World Publisher.

KERNIG, WERNER. 1990 (July). *Automatisches Beweisen von Theoremen über Bilattices.* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

KERNIG, WERNER. 1992 (Apr.). *Modellierung und Verifikation von Switch–Level Spezifikationen mit Hilfe von mehrwertiger Logik.* Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

KNUTH, DONALD E., & BENDIX, P. B. 1970. Simple Word Problems in Universal Algebras. *Pages 263–297 of:* LEECH, J. (ed), *Computational Problems in Abstract Algebras.* Oxford: Pergamon Press.

KREIDLER, MARTIN. 1992 (May). *Implementierung von Dissolution in einen tableau-basierten Theorembeweiser.* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

MOCK, MARKUS. 1990 (Aug.). *Verfahren zur Vereinfachung mehrwertiger Logikfunktionen.* Studienarbeit, Fakultät für Informatik, Universität Karlsruhe.

MORGAN, CHARLES G. 1976. Methods for Automated Theorem Proving in Nonclassical Logics. *IEEE Transactions on Computers*, **C–25**(8), 852–862.

MURRAY, NEIL V., & ROSENTHAL, ERIK. 1986 (Apr.). *Path dissolution for propositional logic.* Tech. rept. TR–86–6. Dept. of Computer Science, SUNY at Albany.

MURRAY, NEIL V., & ROSENTHAL, ERIK. 1987 (July). Path dissolution: a strongly complete rule of inference. *Pages 161–166 of: Proceedings, 6th National Conference on Artificial Intelligence, Seattle.*

MURRAY, NEIL V., & ROSENTHAL, ERIK. 1990a. *DISSOLUTION: Making paths vanish.* Tech. rept. TR–90–?? Dept. of Computer Science, SUNY at Albany.

MURRAY, NEIL V., & ROSENTHAL, ERIK. 1990b. *On the relative merits of path dissolution and the method of analytical tableaux.* Tech. rept. TR–90–5. Dept. of Computer Science, SUNY at Albany.

MURRAY, NEIL V., & ROSENTHAL, ERIK. 1993. Dissolution: Making paths vanish. *Journal of the ACM*, **3**(40), 504–535.

NUTT, WERNER, RÉTY, P., & SMOLKA, GERT. 1989. Basic Narrowing Revisited. *Journal of Symbolic Computation*, **7**(3/4), 295–318.

PELLETIER, FRANCIS JEFFRY. 1986. Seventy-Five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, **2**, 191–216.

PRAWITZ, DAG. 1970. A Proof Procedure with Matrix Reduction. *Pages 207–213 of: LNM 125.* Springer.

REEVES, STEVE V. 1987. Adding Equality to Semantic Tableau. *Journal of Automated Reasoning*, **3**, 225–246.

SCHMITT, PETER H. 1987. *The THOT Theorem Prover.* Tech. rept. TR–87.09.007. IBM Heidelberg Scientific Center.

SCHMITT, PETER H. 1989. *Perspectives in Multi-Valued Logic.* Proceedings, International Scientific Symposium on Natural Language and Logic, Hamburg.

SCHÖPKE, GISELA. 1991 (Oct.). *Möglichkeiten des Einsatzes eines dreiwertigen Theorembeweisers.* IWBS Report 188. Wissenschaftliches Zentrum, IWBS, IBM Deutschland.

SMULLYAN, RAYMOND. 1968. *First-Order Logic.* Springer, New York.

SURMA, STANISŁAW J. 1984. An Algorithm for Axiomatizing Every Finite Logic. *Pages 143–149 of:* RINE, DAVID C. (ed), *Computer Science and Multiple-Valued Logics.* North–Holland, Amsterdam.

WERNECKE, WOLFGANG. 1991. *TCG Language Draft.* Internal Paper.

Index