Automated Reasoning Embedded in Question Answering

Björn Pelzer

Mai 2013

Vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau zur Verleihung des akademischen Grades **Doktor der Naturwissenschaften (Dr. rer. nat.)** genehmigte Dissertation.

Vorsitzender des Promotionsausschusses:	Prof. Dr. Rüdiger Grimm
Vorsitzende der Promotionskommission:	Prof. Dr. Karin Harbusch
1. Berichterstatter:	Prof. Dr. Ulrich Furbach
2. Berichterstatter:	Prof. Dr. Peter Baumgartner

Datum der wissenschaftlichen Aussprache: 03.05.2013

Veröffentlicht als Dissertation an der Universität Koblenz-Landau.

Abstract

This dissertation investigates the usage of theorem provers in automated question answering (QA). QA systems attempt to compute correct answers for questions phrased in a natural language. Commonly they utilize a multitude of methods from computational linguistics and knowledge representation to process the questions and to obtain the answers from extensive knowledge bases. These methods are often syntax-based, and they cannot derive implicit knowledge. Automated theorem provers (ATP) on the other hand can compute logical derivations with millions of inference steps. By integrating a prover into a QA system this reasoning strength could be harnessed to deduce new knowledge from the facts in the knowledge base and thereby improve the QA capabilities. This involves challenges in that the contrary approaches of QA and automated reasoning must be combined: QA methods normally aim for speed and robustness to obtain useful results even from incomplete of faulty data, whereas ATP systems employ logical calculi to derive unambiguous and rigorous proofs. The latter approach is difficult to reconcile with the quantity and the quality of the knowledge bases in QA. The dissertation describes modifications to ATP systems in order to overcome these obstacles. The central example is the theorem prover E-KRHyper which was developed by the author at the Universität Koblenz-Landau. As part of the research work for this dissertation E-KRHyper was embedded into a framework of components for natural language processing, information retrieval and knowledge representation, together forming the QA system LogAnswer. Also presented are additional extensions to the prover implementation and the underlying calculi which go beyond enhancing the reasoning strength of QA systems by giving access to external knowledge sources like web services. These allow the prover to fill gaps in the knowledge during the derivation, or to use external ontologies in other ways, for example for abductive reasoning. While the modifications and extensions detailed in the dissertation are a direct result of adapting an ATP system to QA, some of them can be useful for automated reasoning in general. Evaluation results from experiments and competition participations demonstrate the effectiveness of the methods under discussion.

Zusammenfassung

Die vorliegende Dissertation behandelt den Einsatz von Theorembeweisern innerhalb der automatischen Fragebeantwortung (*question answering* - QA). QA-Systeme versuchen, natürlichsprachliche Fragen korrekt zu beantworten. Sie verwenden eine Vielzahl von Methoden aus der Computerlinguistik und der Wissensrepräsentation, um menschliche Sprache zu verarbeiten und die Antworten aus umfangreichen Wissensbasen zu beziehen. Diese Methoden sind allerdings meist syntaxbasiert und können kein implizites Wissen herleiten. Die Theorembeweiser der automatischen Deduktion dagegen können Folgerungsketten mit Millionen von Inferenzschritten durchführen. Die Integration eines Beweisers in ein QA-System eröffnet die Möglichkeit, aus den Fakten einer Wissensbasis neues Wissen herzuleiten und somit die Fragebeantwortung zu verbessern. Herausforderungen liegen in der Überwindung der gegensätzlichen Herangehensweisen von Fragebeantwortung und Deduktion: Während QA-Methoden normalerweise darauf abzielen, auch mit unvollständigen oder fehlerhaften Daten robust und schnell zu halbwegs annehmbaren Ergebnissen zu kommen, verwenden Theorembeweiser logische Kalküle zur Gewinnung exakter und beweisbarer Resultate. Letzterer Ansatz erweist sich sich aber als schwer vereinbar mit der Quantität und der Qualität der im QA-Bereich üblichen Wissensbestände. Die Dissertation beschreibt Anpassungen von Theorembeweisern zur Überwindung dieser Hürden. Zentrales Beispiel ist der an der Universität Koblenz-Landau entwickelte Beweiser E-KRHyper, der im Rahmen dieser Dissertation in das QA-System Log-Answer integriert worden ist. Außerdem vorgestellt werden zusätzliche Erweiterungsmöglichkeiten auf der Implementierungs- und der Kalkülebene, die sich aus dem praktischen Einsatz bei der Fragebeantwortung ergeben haben, dabei aber generell für Theorembeweiser von Nutzen sein können. Über die reine Deduktionsverbesserung der QA hinausgehend beinhalten diese Erweiterungen auch die Anbindung externer Wissensquellen wie etwa Webdienste, mit denen der Beweiser während des Deduktionsvorgangs gezielt Wissenslücken schließen kann. Zudem ermöglicht dies die Nutzung externer Ontologien beispielsweise zur Abduktion. Evaluationsergebnisse aus eigenen Versuchsreihen und aus Wettbewerben demonstrieren die Effektivität der diskutierten Methoden.

For Ania

Acknowledgements

While this dissertation is the result of several years of my work, this work would have been impossible without the aid and advice by many people I am fortunate to know. Here I would like to express my gratitude towards these supporters.

First of all I thank my thesis supervisor Ulrich Furbach for his guidance and encouragement in my work, and for allowing me the chance at this project in the first place. His tutelage provided me with manifold experience in the world of science, and I am indebted both to his willingness to explore promising areas, and to his pragmatism at not losing sight of the main goals.

Likewise I thank Peter Baumgartner, who let me take my first steps in AI, who also worked with me on various occasions over the years, allowing me to broaden the scope of my experiences, and who took on the task of being the second reviewer of this thesis.

My gratitude goes out to the members of the AGKI for the great work environment. Here I would like to single out Markus Bender for his contributions to LogAnswer, and Beate Körner for always knowing everything and for taking care of bureaucracy. I am also obliged to former AGKI member Christoph Wernhard, whose work provided the foundation for my own.

Regarding the LogAnswer project I am also most thankful to my colleagues in Hagen, notably Ingo Phoenix né Glöckner, for the good cooperation over many years.

My research was backed by the Deutsche Forschungsgemeinschaft (DFG) and of course the Universität Koblenz-Landau, which I appreciate greatly.

I thank my family and friends for their support and patience - in particular my parents Anna-Lena and Heinz-Leo Pelzer, for proofreading and constructive criticism, for always being there for me and for helping out in countless instances. I hope I can make up for this somehow. In the same way I am indebted to Anna Stożek for her moral support and for bearing with me.

For a special kind of inspiration and motivation I am thankful to Iain M. Banks, one of the few authors who have shown AI contributing to a better future.

Finally I am grateful to Hans de Nivelle for bringing CADE to Wrocław in 2011, a decision that has enriched my life in so many ways.

Preliminary Remarks

The work presented in this dissertation is part of the LogAnswer research project which involved several scientists, see Section 5.4. For a proper consideration of my own contributions a broader understanding of LogAnswer is required. As some aspects of LogAnswer were dealt with in cooperative work while others were the responsibility of individuals, generally the *author's "we"* will be used for the sake of consistency throughout the remainder of the dissertation. However, on some occasions I will refer to myself explicitly when clarifying which parts are my own contributions.

Also, some sections of this dissertation describe work, some of it my own, which was not conducted during my research for this thesis. These aspects are therefore not new contributions, but they are nevertheless included for clarity, as my dissertation builds upon them and would be difficult to understand otherwise. This will be clearly mentioned when introducing these subjects.

References to my own publications have numeric labels ([1], [2], ...), whereas other references use mnemonic labels consisting of abbreviations of the authors' names and the year of publication, for example [Wer03].

A collection of materials relevant for this dissertation, including theorem prover implementations and evaluation logs, is available online.[1]

[1] `http://userpages.uni-koblenz.de/~bpelzer/dissertation/materials.tar.gz`

Contents

Chapter 1

Introduction

The field of question answering (QA) deals with the development of computer systems which automatically compute correct and concise answers to questions phrased in a natural language. For example, given the question *"Which planet is closest to the Sun?"*, a QA system should respond with *"Mercury"*. QA has the potential to complement or even replace conventional search engines like Google,[1] as these cannot handle questions in a satisfactory manner. Usually they attempt no semantic analysis of an input question. Instead they regard it as a series of search words and respond with a set of document references, which the users then need to study on their own to see whether there is an answer. In contrast, an ideal QA system is intuitive to use, it delivers exactly what the user is actually looking for, and it is also well-suited to the small screens of compact mobile devices where search engine results become unwieldy. QA has been of interest to artificial intelligence (AI) research for a long time. However, the progress from early QA beginnings in systems like *Baseball* [GWCL61] and *SHRDLU* [Win71] has been slow. Even famous modern examples like the *Watson* system [FBCC+10] that won a television quiz show or the QA features built into some smartphones[2] do not yet have the combination of efficiency and reliable performance that would allow QA systems to gain widespread acceptance.

This is because QA system development is far more difficult than developing search engines, and it draws from more fields of research. The analysis of the questions requires methods from computational linguistics. The creation and maintenance of the large knowledge bases typically employed by QA systems are matters of knowledge representation and knowledge engineering. Answer facts are obtained from these knowledge bases via information retrieval methods, and further natural language processing may be necessary to turn such facts into human readable answers. Many of these steps still involve open research questions, and the difficulties in coping with the enormous quantities of data cannot be ignored.

One problem are the gaps and flaws in the knowledge bases. Whether created manually by knowledge engineers or automatically from textual sources, knowledge bases for QA systems are so large that their contents cannot be scrutinized effectively. Human oversights as well as flaws in the sources or in the automated translation result in a knowledge base that is imperfect and incom-

[1] `http://www.google.com`
[2] `http://www.apple.com/iphone/features/siri.html`

plete. Not every answer will be stored explicitly. Therefore a good QA system must be able to deduce implicit knowledge from the facts in its knowledge base. For example, given the sentence *"The Mont Blanc rises 4,810 meters above sea level."*, human readers should have no problem answering the question *"How high is the Mont Blanc?"*. Indeed, these people would likely even be unaware of the inferences they made on the semantics of *"rises"* and *"high"*, allowing them to link these notions and to identify the answer. Such inferences are not obvious to a QA system, and significantly more complex reasoning will be necessary for a QA system to meet the expectations of the public. In spite of this, for performance reasons QA systems usually eschew such deep inferencing. Instead they rely on "shallow" methods that operate only on the syntactic level.

This is where the topic of this dissertation comes into play. Automated theorem provers (ATP) are the implementations of logical deduction calculi, the subject of automated reasoning (AR). A prover applies the rules of its calculus to a given logical input, and thereby it can deduce new facts and proofs in deep derivations with millions of inference steps. By embedding an automated theorem prover into a QA system this reasoning strength can be utilized for question answering, with the prover deriving implicit answers from the knowledge base. This embedding brings along a number of challenges, as the methodologies of QA and AR are diametrically opposed in some of their goals. QA aims for robustness and speed to obtain useful results from incomplete or faulty data. AR on the other hand requires exact results which are rigorously proven, a precision that renders it brittle towards imperfect data and which thus is hard to reconcile with the massive, flawed knowledge bases of QA and the short response times desired by human QA users. This dissertation will show how both fields can be combined in a way that AR enhances the reasoning strength of QA while preserving robustness. The results are improvements in QA performance and also beneficial extensions to AR. The central example of the dissertation is the automated theorem prover *E-KRHyper* which I developed at the Universität Koblenz-Landau based on earlier ATP systems, and which was embedded into the QA system *LogAnswer* as part of my research work. LogAnswer is intended for arbitrary questions in German language, and it uses a knowledge base derived from the German Wikipedia.[3] The overall development of LogAnswer was joint work[4] between the AGKI[5] of Professor Ulrich Furbach at the Universität Koblenz-Landau and the IICS[6] of Professor Hermann Helbig at the FernUniversität in Hagen.

The idea of using automated deduction in QA is not new. An early attempt was the system presented by Fischer Black in 1968 [Bla68]. This was severely restricted by the limitations of its time, like the lack of parsers, electronic lexicons and digital knowledge sources. It did not actually use natural language, instead it operated on simple logical formulas which had an intuitive resemblance to English sentences, without any actual underlying representational formalism. Its

[3] `http://de.wikipedia.org`

[4] The cooperation project is funded by the DFG (Deutsche Forschungsgemeinschaft, *German Research Foundation*) under the contracts FU 263/12-1, HE 2847/10-1, FU 263/12-2 and GL 682/1-2.

[5] Arbeitsgruppe Künstliche Intelligenz (*Artificial Intelligence Research Group*): `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IFI/AGKI`

[6] Intelligent Information and Communication Systems: `http://pi7.fernuni-hagen.de`

knowledge base was tiny, and the whole system was basically a manually created proof of concept unsuitable to handle much beyond the example questions.

DORIS [Bos01] from 2001 is more advanced in that it uses logic for the semantic analysis of natural language. However, this system is aimed at discourse regarding a limited domain, and it cannot answer arbitrary questions.

FALCON [HMM+00] is an actual QA system with deduction and intended for arbitrary questions. As its knowledge base is derived from a lexical database, it does not have the semantic depth to support complex inferences.

PowerAnswer [MCHM03] is an English language QA system that is similar to LogAnswer in that it also uses Wikipedia and deduction. However, for the most part PowerAnswer operates directly on the textual sources instead of on a formal representation in a knowledge base, again limiting the depth of its reasoning.

A large formal knowledge base is utilized by the QA system *MySentient Answers* [CMB05], built from a subset of the *Cyc* ontology [Len95]. The system is commercial and not intended for arbitrary questions. Instead customers have to provide formally encoded knowledge about their specific application domains.

The Portuguese language system *Senso* [SQ07] is even closer to LogAnswer in its use of a large formal knowledge base and deduction. It employs Prolog [CR93a] for its reasoning, not a full theorem prover, and it is only intended for competitions with generous time limits, not for the short response times allowed for a QA system on the web.

Using a theorem prover is rare in QA. In the QA competition tracks of CLEF[7] from 2008 to 2011 [FPA+08, PFS+09, PFR+10, PHF+11], LogAnswer was the only competitor to employ a full ATP system. Nevertheless some other QA systems have been built around provers. For example, the ATP system *SNARK* [SWC00] is used in a QA component of *Amphion* and *BioDeducta* [WS08]. These are intended as research tools within limited domains. Questions cannot be entered freely, instead a query composition interface ensures that only valid questions are formed. Another example is *SPASS-XDB* [SSW+09, SST+10], a system that connects a number of online knowledge sources, ontologies and databases to a modified version of the theorem prover *SPASS* [WSH+07]. The system has an experimental interface using *Attempto Controlled English* [FSS99], a subset of English intended for knowledge representation. This restriction of the input language requires an experienced user. The overall system is very ATP-centric, without robust natural language processing.

In contrast to the listed systems, the goal of LogAnswer is to use a full automated theorem prover within a QA system with comprehensive natural language processing and robustness. Also, LogAnswer is intended to be scalable to different use cases, from a search engine replacement with response times of at most a few seconds to a research tool which can allow several minutes to process one question. Integrating E-KRHyper in a way that supports this operation has required numerous changes to the prover. Some of these adaptations may be useful for ATP in general. Apart from improving the reasoning capabilities of its host QA system, an embedded theorem prover also adds new possibilities to answering questions, for example by accessing web services during the derivation or by using abduction. All these modifications and extensions will be discussed and when possible evaluated.

[7] Cross-Language Evaluation Forum: `http://www.clef-campaign.org`

After this cursory introduction the dissertation is structured into the following chapters, which will elaborate upon many of the notions mentioned so far only in passing. Each chapter summary here is accompanied by references to relevant own publications when applicable. Naturally these references are also found later on in the actual chapters themselves.

Chapter 2 introduces the formal preliminaries for this dissertation in the form of several basic concepts and notations related to first-order logic. While these are required for an understanding of the later chapters, their usage throughout the dissertation generally conforms to the standards in the field, so readers familiar with this matter may choose to skip ahead to the next chapter.

Chapter 3 provides an overview of automated reasoning with an emphasis on automated theorem proving. Theorem prover implementations will be discussed, as will their evaluation with problem libraries and in competitions.

Chapter 4 is the analogue overview for question answering, a field that still faces many open problems and challenges, but which has already led to numerous experimental implementations and a community which evaluates such systems.

Chapter 5 then details the motivation for the dissertation by discussing the combination of question answering and automated reasoning, identifying respective problems and how each field can contribute to the other. The main challenges are identified, and the LogAnswer project is presented, forming the background of this research work.

Chapter 6 is a precursor to the subsequent system description of E-KRHyper. It summarizes the two hyper tableaux calculi implemented by this prover. Note that these are not new contributions, and a similar summary was part of my diploma thesis [14], but the information and terminology presented in this chapter are essential to an understanding of this dissertation. [1, 2]

Chapter 7 provides an overview of the theorem prover E-KRHyper. Some aspects are described in detail, when they are relevant for adaptations or experiments in later chapters. Note that the implementation of the initial E-KRHyper prototype was the subject of my diploma thesis [14]. However, since then the prover has been reworked extensively as part of my research for this dissertation, including the basic operation. Hence this chapter describes the current matured version of E-KRHyper that is employed in LogAnswer. It supersedes any earlier system descriptions of the prototype E-KRHyper, since those no longer apply to the prover as used in this project. [16]

Chapter 8 is a system description of LogAnswer. The processing chain is explained step by step, from the question of the user to the presentation of the answers. The basic integration of E-KRHyper and its interaction with the overarching systems are described as well. Details of this integration are the focus of subsequent chapters. [6, 7, 15]

Chapter 9 explores fundamental properties which may make automated theorem provers more or less suitable for embedding in a QA system like LogAnswer. The chapter includes a brief system description of the prover E-Darwin. E-KRHyper and E-Darwin then serve as an example of two closely related provers which nevertheless show significant performance differences on reasoning tasks occurring in LogAnswer. We investigate reasons for this and also evaluate other theorem provers. [3, 8]

In Chapter 10 we detail the way E-KRHyper handles indexing and subsumption of clauses with multiple literals. Our method is unique to the best of our knowledge and can be useful for general theorem proving outside QA.

The following Chapter 11 becomes more specific to QA in that it investigates technical adaptations of a theorem prover to the embedding inside a QA system. This includes hardening the implementation against large clause sets and supporting special modes of operations that are useful in the context of QA. [15]

Chapter 12 deals with logic-based means of handling large reasoning problems as encountered in QA. These are primarily axiom selection methods, and we describe both complete and incomplete approaches as implemented in E-KRHyper. [8]

In Chapter 13 we detail the relaxation method used by LogAnswer and E-KRHyper. This adds a degree of robustness to our usage of automated reasoning, enabling LogAnswer to cope with the flaws that are inevitable in a large, automatically generated knowledge base. [8, 10, 15]

While many of the previous chapters contained their own evaluations of particular aspects of E-KRHyper, a comparative evaluation of full QA systems is difficult to achieve internally within a research group, as such systems are hardly portable. Instead QA competitions form the common evaluation venue, and Chapter 14 describes the participation of LogAnswer in the CLEF QA tracks over the years. [9, 11, 12, 13]

As QA competitions have a number of limitations due to their need to provide equal conditions for all participants, they are not very representative for a real-world application of QA. Chapter 15 therefore describes our own additional evaluation of LogAnswer on a large set of questions from an online QA forum. Such questions are more difficult to handle, and in the forum use case it is preferable not to answer at all rather than posting a wrong answer. We explore improved filtering methods for this purpose. [4, 5]

In Chapter 16 we go beyond encyclopedic, static knowledge bases and describe an experimental extension of E-KRHyper by web services which can provide current data. Such a connection introduces technical difficulties like network latency, and an effective integration requires asynchronous communication and reasoning. We develop a sound formal basis for this by extending the hyper tableaux calculi underlying E-KRHyper, and we address the incompleteness inherent to such a connection. While we argue that this problem cannot be solved, we consider different approaches to alleviating it. We also describe the implementation of this extension in E-KRHyper, and we discuss using this connection to access concept hierarchies of externally stored ontologies for the purpose of relaxation by abduction.

In the final Chapter 17 we summarize the findings of this dissertation and explore possibilities for future work.

Chapter 2

Formal Preliminaries

This short chapter introduces the formal preliminaries required for an understanding of the calculi and examples in this thesis. The first part explains basic classical first-order logic (FOL), while the second part describes the addition of equality and several associated concepts. The chapter primarily serves to lay down the terminology as it is used in this thesis, and the definitions and notations generally conform to the standard usage in the field.

2.1 First-Order Logic

The first part summarizes the fundamental concepts of first-order logic, in particular in its clausal form. A basic understanding of FOL is assumed, and as such there will be no detailed explanations and proofs here. The focus is on defining the terminology and the relations between the concepts.

2.1.1 Terms and Substitutions

The first-order language used in the following is built upon a fixed signature $\Sigma = (\mathcal{F}, \mathcal{P})$, with $\mathcal{F}$ being a set of *function* symbols and $\mathcal{P}$ a set of *predicate* symbols. Both sets are infinite and disjoint. Each symbol has a fixed arity. A function symbol with arity zero is called a *constant*. A third set $\mathcal{X}$ is the set of variables, which is infinite and disjoint from $\mathcal{F}$ and $\mathcal{P}$. This allows the definition of the set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$:

1. Every constant $c \in \mathcal{F}$ and every variable $x \in \mathcal{X}$ is a term.
2. If $t_1, \ldots t_n$ are terms and $f \in \mathcal{F}$ is a function symbol with arity n, then $f(t_1, \ldots t_n)$ is a term.

A term s is a *subterm* of a term t with $t = f(t_1, \ldots t_n)$ for some function symbol f with arity n, if $s = t$ or if $s = t_i$ or if s is a subterm of t_i, for some $1 \leq i \leq n$. s is a *proper subterm* of t if s is a subterm of t and $s \neq t$. $vars(t)$ denotes the set of variables of a term t. A term t is called *ground* if $vars(t) = \emptyset$.

A *substitution* σ is a mapping from $\mathcal{X}$ to $\mathcal{T}$, with a finite *domain* $dom(\sigma) = \{x \mid x\sigma \neq x\}$ and a finite *range* $ran(\sigma) = \{x\sigma \mid x\sigma \neq x\}$, $x \in \mathcal{X}$. A *ground substitution* γ is a substitution with $vars(ran(\gamma)) = \emptyset$. A *renaming* ρ is a substitution which is a bijection of $\mathcal{X}$ onto itself. Given two terms s and t,

a substitution σ is a *unifier* for s and t if $s\sigma = t\sigma$. The composition of two substitutions σ and λ is written as $\sigma\lambda$. σ is a *most general unifier* (*mgu*), if for any other unifier τ for s and t there is a substitution λ with $\sigma\lambda = \tau$.

A term s is an *instance* of a term t (written as $s \gtrsim t$) if there is a substitution σ such that $s\sigma = t$, and t is then called a *generalization* of s, and we also write that t *subsumes* s, and s *is subsumed by* t. s is a *variant* of t (written as $s \sim t$) if there is a renaming ρ such that $s\rho = t$.

If $t_1, \ldots t_n$ are terms and $p \in \mathcal{P}$ is a predicate symbol with arity n, then $p(t_1, \ldots t_n)$ is an *atom*. A *literal* is an atom or the negation ($\neg$) of an atom. A literal is ground if its component terms are ground. $\overline{L}$ denotes the complement of L. The notions of substitutions, renamings, instances, generalizations and variants as well as subsumption are extended to atoms and literals in the obvious way.

2.1.2 Formulas

The set of (well-formed) formulas has the following inductive definition:

1. An atom A is a formula.
2. If F is a formula, then its negation $\neg F$ is a formula.
3. If F and G are formulas, then the conjunction $F \wedge G$ and the disjunction $F \vee G$ are formulas.
4. If F is a formula and x is a variable, then $\exists x F$ and $\forall x F$ are formulas.

The binary connectives $\leftarrow$, $\rightarrow$ and $\leftrightarrow$ are also used as the common shorthand notations for implications and the biconditional. The truth constants $\top$ (true) and $\bot$ (false) may be used as well (see Section 2.1.3).

A set of formulas may sometimes be referred to as a *theory*.[1] The formulas of a theory are referred to as *axioms*.

2.1.3 Satisfiability

A *(Herbrand) interpretation* I is the set of ground atoms that are true in I. I is a *model* for a formula F ($I \models F$) if F is true in I following the usual semantics of FOL. F is *satisfiable* if it has a model, and it is *valid* (or a *tautology*) if it is true in any interpretation, also written as $\models F$. $\top$ represents an unconditionally true atomic formula. Conversely, F is *unsatisfiable* if it has no model ($\not\models F$), and F is *invalid* if it is not valid, i.e. some interpretation I is not a model for F ($I \not\models F$). $\bot$ represents an unconditionally false atomic formula. A formula G is a logical consequence of F if $I \models G$ for every I with $I \models F$. Two formulas F and G are *equisatisfiable* if either both are satisfiable or both are unsatisfiable.

[1] Some authors, see for example [Sch95], require that a theory also includes all of its logical consequences (see Section 2.1.3). However, this view is impractical for the purposes of this dissertation, where often a clear distinction between axioms and consequences must be made. Also, while technically any random set of formulas could be called a theory using the less strict definition in this dissertation, usage of the term will be reserved for sets that attempt to formalize theories in the scientific sense, for example mathematical theories or abstractions of empirical observations. As such the term "theory" is here a somewhat informal way of taking into account a semantic component.

All these notions are extended to sets of formulas (and hence to theories) in the obvious way. A set of formulas is implicitly regarded as a conjunction of these formulas. A formula F is a *theorem* of a theory T if $T \models F$. A theory that is satisfiable is sometimes called *consistent*, and it is *inconsistent* if it is unsatisfiable.

2.1.4 Multisets

A *multiset* is a function M from a set A to $\mathbb{N}$, the set of natural numbers. An element $a \in A$ is an element of M if $M(a) > 0$. Loosely speaking, a multiset is similar to a set, but it may contain multiple occurrences of the same element, and $M(a)$ gives the number of occurrences of a in M. Given two multisets M_1 and M_2, the following functions on multisets are defined:

union: $(M_1 \cup M_2)(x) = M_1(x) + M_2(x)$,

intersection: $(M_1 \cap M_2)(x) = min(M_1(x), M_2(x))$,

difference: $(M_1 \backslash M_2)(x) = max(0, M_1(x) - M_2(x))$,

subset: $M_1 \subseteq M_2 \Leftrightarrow \forall x \in M_1 : M_1(x) \leq M_2(x)$.

2.1.5 Clauses

A *clause* C in clause normal form (CNF) is a multiset of literals. A clause is usually written as a disjunction $A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n$ with $m, n \geq 0$, or in the equivalent multiset notation $\{A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n\}$. It can also be written as an implication $A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$, or the equivalent $B_1, \ldots, B_n \rightarrow A_1, \ldots, A_m$. The multiset $\mathcal{A} = \{A_1, \ldots, A_m\}$ is called the *head* of the clause C while $\mathcal{B} = \{B_1, \ldots, B_n\}$ is called the *body* of C. Accordingly, $A_1, \ldots, A_m$ denote both the *head atoms* and the *head literals* of C, while $B_1, \ldots, B_n$ refer to the *body atoms* and $\neg B_1, \ldots, \neg B_n$ to the *body literals*. The notation $A, \mathcal{A} \leftarrow B, \mathcal{B}$ refers to the clause with the head atoms $\{A\} \cup \mathcal{A}$ and the body atoms $\{B\} \cup \mathcal{B}$. The notions of substitutions, renamings, instances, generalizations, variants and satisfiability are extended to clauses in the obvious way. A formula F can be transformed into an equisatisfiable clause C and vice versa. A clause C *subsumes* a clause D if $C\sigma \subseteq D$ for some substitution σ. If $C\sigma \subset D$, then this is also referred to as *proper subsumption*.

A *Horn clause* is a clause with at most one head literal. A *unit* is a clause consisting of exactly one literal. A clause is *empty* if both its head and its body are empty. The empty clause will be written as $\Box$, it cannot be satisfied. A clause is ground if its literals are ground. Any variables in a clause are taken to be universally quantified. A clause is *range-restricted* if all of its head variables also occur in the body, i.e. if $vars(\mathcal{A}) \subseteq vars(\mathcal{B})$. A clause is called *pure* if none of its distinct head literals share variables, i.e. if $vars(A_i) \cap vars(A_j) = \emptyset$ for $1 \leq i, j \leq m$ and $i \neq j$. A substitution π is a *purifying substitution* for C if $C\pi$ is pure.

A *clause set* is a conjunction of clauses, analogous to sets of formulas. Likewise, a set of clauses may sometimes also be referred to as a theory.

2.1.6 Calculi

A *calculus* for first-order logic is a set of *inference rules* that syntactically transform formulas or clauses into other formulas or clauses. $F \Rightarrow_R G$ denotes that formula F (the *premise*) can be transformed into formula G (the *conclusion*) using the inference rule R of a given calculus. G will also be referred to as having been *derived* from F. Generally a *derivation* is a sequence of inference applications where all premises are taken from the initial input or have been derived in a previous step. However, note that the concept of the derivation must be defined specifically for each calculus. $F \vdash G$ denotes that G results from a derivation starting with F, i.e. G is *provable* from F, and the derivation is a *proof* for F. These notions are extended to clauses as well as to sets of formulas or clauses.

A derivation resulting in $\mathcal{F} \vdash \bot$ for a set of formulas $\mathcal{F}$ (or $\mathcal{C} \vdash \square$ for a set of clauses $\mathcal{C}$) is called a *refutation*. A calculus is *sound* if it only derives logical consequences, i.e. $\mathcal{F} \vdash F$ implies $\mathcal{F} \models F$. This also means that a refutation is only derived for unsatisfiable input, a property sometimes referred to as *refutational soundness*. A calculus is *complete* if it derives all logical consequences, i.e. $\mathcal{F} \models F$ implies $\mathcal{F} \vdash F$. This also means that unsatisfiable input will result in a refutation, a property sometimes referred to as *refutational completeness*. A calculus is *proof confluent* if it is always possible to derive a refutation for unsatisfiable input, regardless of the order of inference applications.

2.2 Equality

Equality is an extension to first-order logic. When formulating theories it is often desirable to express the semantic equality between two terms, thereby establishing their common identity, such that both terms can be used interchangeably, or one can replace the other entirely. Basic FOL certainly allows using the equality symbol as a binary predicate on two terms. To avoid confusion with the ubiquitous symbol "=", the symbol for the equality predicate will be "$\simeq$". Unlike other predicates it will use infix notation, for example $a \simeq b$. Unfortunately, simply introducing this predicate does not automatically capture the intended semantics. Technically this can be achieved without extending FOL, by using the workaround of adding the *equality axioms* to a given theory:

1. *reflexivity:* $\forall x(x = x)$,
2. *symmetry:* $\forall x \forall y(x \simeq y \rightarrow y \simeq x)$,
3. *transitivity:* $\forall x \forall y \forall z(x \simeq y \wedge y \simeq z \rightarrow x \simeq z)$,
4. *function substitutions:* $\forall x \forall y(x \simeq y \rightarrow f(\dots, x, \dots) \simeq f(\dots, y, \dots))$, for every function symbol f,
5. *predicate substitutions:* $\forall x \forall y(x \simeq y \wedge p(\dots, x, \dots) \rightarrow p(\dots, y, \dots))$, for every predicate symbol p.

While this solution has the advantage of allowing even basic FOL systems to reason with equality, it easily leads to an explosive growth in the number of formulas, making this method inefficient for many reasoning problems.

A better way is to embed the handling of equations on the calculus level by using special inference rules which take the semantics of equations into account when encountering the equality predicate. For the purpose of defining such calculi a number of general concepts has become established in the field; these will be introduced in the following sections.

2.2.1 Equations

The (semantic) equality of two terms s and t is denoted by $s \simeq t$, where $\simeq$ is a special predicate symbol with binary arity. An atom $s \simeq t$ is called an *equation*. Equations are symmetric; if $s \simeq t$ holds, then so does $t \simeq s$. The negation of an equation, i.e. $\neg(s \simeq t)$, can also be written as $s \not\simeq t$. An equation of the form $t \simeq t$ or $t \not\simeq t$ is called *trivial*.

When describing a calculus for FOL with equality it is common to regard the equality symbol as the only predicate symbol, while the other predicate symbols are treated like function symbols. This has the advantage of greatly simplifying the specification of the inference rules and the proving of their soundness and completeness. For this purpose a special constant $\mathbf{t}$ is introduced, and any normal atom $p(t_1, \ldots, t_n)$ is then treated as an equational atom $p(t_1, \ldots, t_n) \simeq \mathbf{t}$. Despite this convention, when giving examples the transformed atoms are usually written in the standard way, not as equations, in order to enhance legibility. Since all atoms are equations and there are no non-equational predicates, the Herbrand interpretation is a set of ground equations. The semantics of the equational symbol is covered by extending the notion of interpretations to that of an *E-interpretation*, which is both an interpretation and a congruence relation on its ground terms. Accordingly, the notions of satisfiability are extended to those of *E-satisfiability*, and the logical consequence with equality is written as $\models_{\mathrm{E}}$, although when it is clear from the context, the normal designations will be used.

2.2.2 Positions

As of Section 2.1.1 a term may be composed of multiple subterms. When using equations in reasoning it is often necessary to refer to a specific subterm within a term. This is accomplished by the concept of *positions*. A position is a sequence of natural numbers. If t is a term and p is a position, then $t|_p$ denotes the subterm of t at position p. In particular, if ϵ is the empty sequence and $t = f(t_1, \ldots, t_n)$, then $t|_\epsilon = t$ and $t|_{i.p} = t_i|_p$ for $1 \leq i \leq n$. If p is a position in t, then the notation $t\,[s]_p$ will be used for $t|_p = s$, and $t\,[p/s']$ represents the term obtained by replacing $t|_p$ with s' at position p in t. If p is obvious or unimportant within the context, it can be omitted, so that $t\,[s]$ denotes the term t with the subterm s, and $t\,[s']$ denotes the same term t except for its subterm s having been replaced by s'.

2.2.3 Term Ordering

First-order logic reasoning with equality also often needs to compare terms using a *reduction ordering* $\succ$, i.e. an ordering meeting the following conditions:

1. $\succ$ is a strict partial ordering (irreflexive, antisymmetric and transitive),
2. $\succ$ is well-founded,
3. $\succ$ is closed under context; if $s \succ s'$ for two terms s and s', then $t\,[s]\,|_p \succ t\,[s']\,|_p$ for any term t and any position p in t,
4. $\succ$ is liftable; if $s \succ t$ for two terms s and t, then $s\sigma \succ t\sigma$ for any substitution σ, and finally
5. $\succ$ is total on ground terms.

$\succ$ induces the non-strict ordering $\succeq$. The negated forms are $s \not\succ t$ and $s \not\succeq t$, respectively. Two terms s and t will be called *incomparable*, if $s \not\succ t$ and $t \not\succeq s$. Furthermore, $\succ$ is lifted to atoms, literals and clauses.

Several different reduction orderings exist. The specific definition of an ordering is not important when formalizing inference rules, as long as it is a reduction ordering. However, different reduction orderings may have different advantages and drawbacks for an implementation.

One example of a reduction ordering is the *recursive path ordering* (RPO) [Der82]. Its definition requires a few preceding concepts. First, let $\succ^{mul}$ be the multiset extension of $\succ$: $M \succ^{mul} N$ holds if $N \neq M$ and for all $n \in N \backslash M$ there exists an $m \in M \backslash N$ with $m \succ n$. Also necessary is a precedence ordering $>$, which is a strict order on the set of signature symbols $\Sigma = (\mathcal{F}, \mathcal{P})$.

Now the RPO $\succ_{rpo}$ can be defined. $s \succ_{rpo} t$ holds for two terms s and t, if

1. $t \in \mathit{vars}(s)$ and $s \neq t$, or
2. $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$ with
   (a) $s_i \succ_{rpo} t$ for some $1 \leq i \leq m$, or
   (b) $f > g$ and $s \succ_{rpo} t_j$ for all $1 \leq j \leq n$, or
   (c) $f = g$ and $\{s_1, \ldots, s_m\} \succ_{rpo}^{mul} \{t_1, \ldots, t_n\}$.

Reduction orderings can be extended to atoms, literals and clauses.

An equation $l \simeq r$ is *oriented equation* if $l \succ r$, and it is *orientable equation* if l and r are not incomparable. The special symbol $\mathbf{t}$ introduced for the equational notation of non-equational atoms in Section 2.2.1 is usually assumed to be the smallest symbol in the precedence ordering, and therefore the $\mathbf{t}$ constant is also the smallest term in the term ordering. This ensures that these notational equations based on non-equational atoms are always orientable, and the term representing the original atom will always form the left, larger side of the equation.

2.2.4 Rewrite Systems

Reasoning with equality introduces the possibility to replace terms and subterms with other terms, going beyond substitutions which only affect variables. A helpful notion in this context is that of a *rewrite rule*, which takes the form $l \to r$ for two terms l and r. Unlike an equation a rewrite rule expresses a direction for its term replacement. A set R of rewrite rules is a *rewrite system*. $\to^*$ is the transitive closure of $\to$, and $(l \to^* r) \in R$ denotes that $(l \to r) \in R$ or $\{(l \to t_1), (t_1 \to t_2), \ldots, (t_n \to r)\} \subseteq R$ for some terms $t_1, \ldots, t_n$ with $n \geq 1$. A rewrite system R is ground if l and r are ground for every $(l \to r) \in R$. R is *ordered by the reduction ordering* $\succ$ if $l \succ r$ for every $(l \to r) \in R$. This dissertation will always assume rewrite systems to be ground and ordered. A term t is *reducible by* $l \to r$ if $l \succ r$ and $t|_p = l$ for some position p. t is *reducible with respect to* R if it is reducible by some rule in R, and t is *irreducible with respect to* R if it is not reducible by any rule in R. A rewrite system R is *left-reduced* if l is irreducible with respect to $R \setminus \{l \to r\}$ for every rule $(l \to r) \in R$. R is *fully reduced* if it is left-reduced and r is irreducible with respect to $R \setminus \{l \to r\}$ for every rule $(l \to r) \in R$. R is confluent if for every pair $(l \to^* t_1) \in R$ and $(l \to^* t_2) \in R$ there is also $(t_1 \to^* r) \in R$ and $(t_2 \to^* r) \in R$. R is *terminating* if there is no infinite chain of rewrite rules $(l_1 \to r_1), (l_2 [r_1] \to r_2), (l_3 [r_2] \to r_3), \ldots$ in R. R is *convergent* if it is confluent and terminating.

Chapter 3

Automated Reasoning

Automated reasoning (AR) is the field of artificial intelligence (AI) research which explores methods of reasoning, in particular the formalization of such methods with the objective of making them applicable by computer systems. A driving motivation behind research in AR has been the wish to find mathematical proofs automatically. As a result the development of classical logical and mathematical calculi has taken centre stage within AR, but other aspects of reasoning also receive attention, for example abductive reasoning, default reasoning, probabilistic reasoning, and qualitative and quantitative spatial and temporal reasoning. The many facets of AR are covered by various implementations of automated reasoners, ranging from task-specific algorithms within larger applications to stand-alone reasoners operating on formal problem specifications.

3.1 Automated Theorem Proving

Automated theorem proving (ATP) is the subfield of AR which deals with the implementation and application of automated systems dedicated to reasoning tasks. Throughout this dissertation the concrete reasoning tasks will be referred to as *(reasoning) problems*. Regarding AR in general, a reasoning problem may be defined like this:

Definition 3.1 (Reasoning Problem and Solving)**.** *Given a logic system S consisting of a formal system $F = (\Sigma, L, A, R)$ (composed of an alphabet Σ, a formal language L, an axiom set A and the inference rules R) and associated semantics, a* (reasoning) problem *P is a set of statements in L.* Solving *P means deciding whether P follows from A under R.*

Problems can have many different forms, in particular considering that provers exist for many different logics. As this dissertation primarily deals with first-order logic, we will define the notion of a reasoning problem within the context of FOL (see Chapter 2):

Definition 3.2 (First-Order Logic Reasoning Problem and Solving)**.** *A (FOL reasoning) problem $P = (Ax, Con)$ consists of two possibly empty sets of clauses or formulas, the axiom set (or theory) Ax and the conjecture set Con.* Solving

P means determining whether Con is a theorem of Ax ($Ax \models Con$) or not ($Ax \not\models Con$).

Testing an axiom set Ax for consistency can be seen as trying to solve the reasoning problem $P = (Ax, \{\})$ as even a vacuous conjecture will trivially follow from an inconsistent theory.

The problem concept is used to further define the notion of the systems researched in ATP, the *automated theorem provers*:

Definition 3.3 (Automated Theorem Prover)**.** *An* automated theorem prover *is an algorithm*[1] *which implements a sound calculus and applies it to reasoning problems in order to solve them, without requiring interaction with a human user.*

Note that while the calculus is required to be sound, it does not necessarily have to be complete. If a calculus is unsound, the results of the prover can never be trusted. On the other hand, an incomplete yet sound system still delivers reliable results. Forgoing or preserving completeness is a design decision: For example, an incomplete system may have to give up more frequently than a complete one, but at the same time it may be significantly faster on the problems it can solve, to the degree that it can solve more problems within a given time limit than if it were complete. The ability to operate without human guidance distinguishes ATP systems from the related interactive theorem provers, though in practice a given theorem prover may offer both interactive and fully automatic modes.

The distinction between the axioms and the conjecture is not strictly necessary, but it reflects how reasoning problems are usually presented and handled in practice. Ultimately FOL problems can be expressed as problems of satisfiability: A conjecture Con is a theorem of Ax ($Ax \models Con$) if the negated conjecture is unsatisfiable in combination with the axioms ($Ax \wedge \neg Con$). With an empty conjecture the theory Ax is consistent if it is satisfiable, and it is inconsistent if it is unsatisfiable. Given the refutational nature of many FOL calculi like resolution [Rob65a] and superposition [NR95], it is common practice for provers to turn a given problem $P = (Ax, Con)$ into an equivalent satisfiability problem $P^{sat} = (Ax^{sat}, \{\})$ with $Ax^{sat} = Ax \cup \neg Con$. The distinction between axioms and conjecture becomes irrelevant at this point, and following common usage in the ATP community we identify P^{sat} with its combined "axiom" set Ax^{sat}, which allows us to call P^{sat} satisfiable or unsatisfiable. The prover then attempts to prove P^{sat} unsatisfiable by means of a refutational proof.

Some provers may also be able to recognize if a problem turns out to be satisfiable, and they can try to produce a model in such cases. This means that some ATP systems are effectively trying to *decide* the satisfiability of problems, a task that is not guaranteed to yield a definitive result given the general undecidability of many logics. In the sequel, when it is not necessary to distinguish between different problem types and results, a prover will simply be referred to as having *solved* a (reasoning) problem P once it has correctly decided the satisfiability of P^{sat}. Such a decision will be called a *result*, and it is sufficient for a result to consist of a compact statement regarding the satisfiability of the given problem (for example *"satisfiable"* or *"unsatisfiable"*). A proof is not part

[1] Note that we use the notion of *algorithm* common in the ATP field, which means that it is a procedure which is not necessarily terminating.

of the result, and a prover is not required to be able to produce proofs. However, the ability to provide proofs certainly makes a prover more useful, and for most modern FOL provers (see below) a proof will accompany the result.

Despite their limitations, automated theorem provers successfully solve difficult reasoning problems on a regular basis. An early such case which brought ATP into the public limelight was the prover *EQP* finding a proof for the Robbins Conjecture in 1996 [McC97], a problem that had been eluding mathematicians since 1933. With their logic-based operation ATP systems lend themselves to mathematical problems and tasks like verification and model checking, but their application domains can be expanded by suitable translations into formal logic.

Several classes of automated theorem provers can be identified. Those provers restricted to propositional logic are referred to as SAT solvers. Usually they implement the DPLL (Davis-Putnam-Logemann-Lovelace) algorithm [DP60] in some manner. *MiniSat* [ES03] is an example of a SAT solver.

There is a larger variety of calculi among ATP systems for first-order logic. Resolution-based provers include *Otter* [McC03], *SPASS* [WSH$^+$07] and *Vampire* [RV02]. Others work with instance-based methods, for example *iProver* [Kor08] and *Darwin* [BFT06], or they operate on various forms of tableaux, like *SETHEO* [MIL$^+$97] and *KRHyper* [Wer03]. Many of these also integrate some form of equality handling, often by superposition [BG98], or they have a specific equational prover variant for this purpose, keeping the purely first-order version unburdened by the overhead required for equational reasoning. On the converse there are also purely equational provers, for example the superposition-based *E* [Sch02], and also *Waldmeister* [Hil03], which specializes in unit equational problems. First-order logic introduces a number of implementational requirements which increase the complexity of such theorem provers compared to SAT solvers: efficient term indexing is needed for the storage of compound terms with variables, inference operations like subsumption and unification necessitate variable substitution algorithms, and redundancy criteria using concepts like term weight and term ordering help making the often infinite Herbrand universe manageable.

SMT solvers (*Satisfiability Modulo Theories*) are first-order automated theorem provers which extend logic with other theories, for example numerical or set theories, expanding the expressivity of their problem specification languages with special evaluable symbols. SMT solvers find use in hardware and software verification due to their ability to reason about data structures. One such prover is *Yices* [DdM06], which is used in the higher-order logic (HOL) prover and verification tool *Isabelle* [EM08].

HOL provers implement predicate variables and lambda notation to reason about functions and predicates. They are rarely fully automatic, and as such most of them must be regarded as interactive theorem provers rather than ATP systems. However, automated HOL provers do exist, for example *LEO-II* [BPT08].

The lines between the different types of ATP systems can be blurred. Some FOL theorem provers have logic extensions similar to those of SMT solvers, but they tend to be afterthoughts rather than being based on fully fledged background theories. Meta provers employ other provers for specific problem types. An example of this is the aforementioned LEO-II, which breaks down HOL problems into FOL subproblems that can then be solved by the ATP systems

E and SPASS. FOL provers may use a similar approach, like the iProver system mentioned above, which employs the SAT solver MiniSat for propositional problems.

First-order system development achieved a dominating position in ATP research early on, and while other branches are catching up, FOL proving remains one of the best developed areas in ATP. This is no surprise: On the one hand propositional logic is subsumed by FOL, and its expressivity can be too limited for practical applications when used on its own. SMT and higher-order logic on the other hand build upon FOL methods, but their many different manifestations and semantics mean that there is little common ground for their implementations, although in recent years there has been some success at standardization in SMT (see Section 3.3). In contrast, first-order ATP systems can all operate on the same logical problems (modulo minor parser or syntax adaptations). This allows the creation of benchmark suites which greatly simplify performance comparisons between systems. Also, implementation techniques and problem transformations can be adopted by many theorem provers, leading to a general improvement of the field.

Given this prominence of first-order theorem provers within ATP and owing to the fact that systems of this type were developed and used within the LogAnswer project, the remainder of this dissertation will focus on FOL systems when referring to theorem provers and ATP research.

3.2 Theorem Prover Implementation

Later chapters will discuss several theorem prover implementations in detail, so this introduction will only provide a very cursory overview.

The development of automated theorem provers is characterized by the undecidability of first-order logic. Any naive approach at automatically solving logic problems will quickly become entangled in a combinatorial explosion of inference possibilities. A first step towards handling this issue lies in using a clause normal form (CNF) representation of the problems. This standardized form lends itself to the specification and implementation of machine-oriented reasoning calculi, and it is used in most calculi and provers with few exceptions (see below).

It is also found in the programming language *Prolog* [CR93b], which can be seen as a very basic form of automated deduction, as it offers functionality somewhat comparable to theorem provers. Its mostly declarative language constructs allow theory specifications which are close to first-order logic, with Prolog programs being written in the form of clauses. Features like arithmetic evaluation and default negation even enable Prolog to approximate SMT-solving to a degree. However, Prolog is effectively limited to Horn-logic, which poses a significant restriction to its expressiveness. Also, an internal database maintains the clauses in Prolog, and while this relieves the programmer of constructing data structures for clause storage and search methods for inference partners, it is effectively a 'black box' and thus not very adaptable to optimizations. Despite the relation to logic, Prolog constructs are not entirely declarative; for example, the ordering of the clauses in a Prolog program can have an effect on the result of its execution, and imperative database operations like `assert/1` and `retract/1` allow side effects within Prolog. The standard unification algorithm

used throughout Prolog has no *occurs check*, meaning that a variable x may unify with a term containing x, leading to a cyclic substitution and possibly unsoundness.[2] The unbounded depth-first search method of Prolog can run into cycles even for problems that should be trivial to prove. For example, consider the following simple Prolog program:

```
1: p(X):-p(X).
2: p(a).
```

Given this program and the query `p(X).`, Prolog will loop by cyclically chaining backward over the clause in line 1, never finding the answer in line 2 as the search order always prefers the first clause.

Although there are limitations, Prolog is nevertheless an important tool in automated deduction. The *PTTP* (Prolog Technology Theorem Prover) [Sti87] system modifies a Prolog compiler to allow sound reasoning on full first-order logic. *PROTEIN* [BFK94] and the aforementioned SETHEO are provers based on PTTP. *SATCHMO* [MB88] takes a different approach from PTTP: It is based on a model-generating calculus which is not affected by the weaknesses of Prolog, and thus a compact high-level implementation in Prolog is sufficient for SATCHMO, avoiding the need for compiler modifications. *leanTaP* [BP95] and *leanCoP* [OB03], the former based on semantic tableaux and the latter on the connection calculus [Bib81], are even more compact Prolog-based theorem provers. Due to their small size (one version of leanCoP has only eight lines of code) they can be seen as attempts at extending Prolog to full FOL with minimal means. This illustrates the usefulness of Prolog as a language for prototype systems, as new ideas can be tested with little implementation work.

The reasoning method of Prolog is notable for operating in a "top-down" fashion, starting with a query (equivalent to the conjecture in ATP) and then reducing it to the axioms. While this goal-oriented approach has the potential to avoid some irrelevant inferences (see Section 12.2.1), most modern theorem provers, including systems based on Prolog, have adopted the opposite direction. They are *saturation-based* theorem provers, which start with the axioms and continuously derive new clauses, until they derive the conjecture, which in combination with the negated conjecture (see Section 3.1) leads to a refutational proof. This "bottom-up" approach, while less goal-oriented, has the advantage of allowing powerful redundancy criteria that can remove clauses or prevent their derivation without affecting soundness and completeness. Saturation-based theorem proving has thereby gained a sufficient edge that it is used by the vast majority of state-of-the-art ATP systems, including the most successful provers.

Besides going for this generally more successful strategy, most modern theorem provers are highly optimized, with all aspects of their implementations tightly integrated. Modularization is a hindrance at the desired level of performance, and typically a compiled ATP system is a compact, monolithic executable. In the calculus the inferences and the redundancy criteria are expressed as neatly delineated rules, but inside the impementation they are often difficult to recognize, sometimes being interlocked and sometimes broken down into scattered substeps, all with the goal of saving some amount of work and processing time. The disadvantage to this is that it is difficult to maintain a theorem

[2] Prolog has a remedy in the form of the `unify_with_occurs_check/2` predicate, but the programmer has to be aware of the problem and explicitly use this sound yet less efficient operator whenever there is a risk of unsound unifications.

ATP	calculus	KLOC	lang.	E	P	M	NF
Ayane 2 [Wal]	sup.	2	C#	✓	✓	-	✓
Darwin 1.4.5 [BFT04]	inst.	55	OCaml	-	✓	✓	-
E 1.1 [Sch02]	sup.	178	C	✓	✓	-	✓
E-Darwin 1.3 [3]	inst.	70	OCaml	✓	✓	✓	✓
E-KRHyper 1.1.4 [16]	tab.	48	OCaml	✓	✓	✓	✓
Geo 2010C [dNM06]	res.	31	C++	✓	✓	✓	✓
iProver 0.7 [Kor08]	inst.	30	OCaml	-	✓	✓	-
iProver-Eq 0.6 [KS10]	inst.	94	OCaml	✓	✓	✓	-
leanCoP 2.2 [OB03]	con.	1	Prolog	-	✓	-	✓
Metis 2.2 [Hur03]	res.	25	SML	✓	✓	✓	✓
Muscadet 4.0 [Pas01]	n.d.	3	Prolog	✓	✓	-	✓
omkbTT 1.0 [WM10]	comp.	19	OCaml	✓	✓	-	n/a
Otter 3.3 [McC03]	res.	43	C	✓	✓	-	✓
Paradox 3.0 [Cla03]	inst.	13	Haskell	✓	-	✓	✓
Vampire 10.0 [RV02]	res.	130	C++	✓	✓	-	✓
Vampire 11.0 [RV02]	res.	61	C++	✓	✓	-	✓
Vampire 0.6 [RV02]	res.	94	C++	✓	✓	-	✓
Waldmeister C09a [Hil03]	comp.	79	C	✓	✓	-	n/a
Waldmeister 710 [Hil03]	comp.	79	C	✓	✓	-	n/a
Zenon 0.6.2 [BDD07]	tab.	21	OCaml	✓	✓	✓	✓

Table 3.1: First-order ATP systems participating in CASC 2010. Column explanations: *ATP* - prover name and version; *calculus* - general calculus type; *KLOC* - lines of code measure (in thousands); *lang.* - programming language; *E* - built-in equality handling; *P* - capable of proof output; *M* - capable of model output; *NF* - built-in normal form transformation. Calculus abbreviations: *comp.* - completion; *con.* - connection calculus; *inst.* - instance-based; *n.d.* - natural deduction; *res.* - resolution; *sup.* - superposition; *tab.* - tableaux. See the text for further explanation.

prover and to introduce new features. The operation of the entire system must be kept in mind during any modification, and as a result the work on a theorem prover is often the responsibility of a lone developer who has to keep a complete and detailed overview of the implementation. Therefore it is not uncommon that planned extensions to a prover or personnel changes in its developer group lead to the reimplementation from scratch. This may be simpler than adapting the original code to use cases it was not intended for, respectively having new developers familiarizing themselves with the complex original. Basic ideas and the underlying calculus then form the only common basis between major release versions of the same prover.

An overview of ATP systems is given in Table 3.1, which provides information about the 20 unique FOL theorem provers that participated in the ATP-competition CASC [PSS02, SS06] of 2010[3] (for more details on the CASC, see the following Section 3.3). Not listed are the participating higher-order systems, nor systems which are only minor variants of a listed prover. The table lists the

[3]The 2010 competition was chosen for this comparison, because the 2011 competition had considerably less participants, essentially featuring a subset of the listed provers.

calculi implemented by the systems. Only the general calculus family is indicated, and there may still be significant differences between two calculi of the same family. For example, while both Zenon and E-KRHyper are tableaux systems, the Zenon system implements an analytic tableaux calculus with free variables, whereas E-KRHyper uses the hyper tableaux calculus [BFN96][1] without free variables. Instance-based methods are implemented by several systems, for example the Inst-Gen calculus in iProver and the Model Evolution calculus [BT03] in Darwin. Muscadet is the only system that is not clearly saturation-based, it is based on natural deduction [Ble71]. Provers can also use aspects of multiple calculi: For example, several systems enhance their calculus with inference rules adapted from superposition in order to handle equality; in such cases only the basic calculus is stated in the table.

The *KLOC*-column shows how many thousand lines of code comprise each system. These numbers were determined from the source code files in the crudest fashion, without consideration for actual complexity, coding conventions, comment lines and whitespace. While this measure is highly inexact, it does provide a general idea of the scale of the implementations, which amount to approximately 53,000 lines of code on average.

The fourth column shows which programming language is used for the bulk of each system. Many provers also have a few scripts for installation or testing purposes, written in some scripting language like Perl or Python, but this is not listed here. There is no clear favorite language. C [KR78] and its derivatives C++ [Str86] and C# [HWG03] are used for nine systems (some of which are closely related), which is not too surprising, as these languages have gained widespread adoption due to their efficiency and maturity. More noteworthy is that no less than seven provers are written in OCaml[4], a language that does not enjoy the general prevalence of the C-family. However, as a strongly typed functional-imperative language with capable compilers for most operating systems and many architectures, OCaml has found acceptance in the automated reasoning community since shortly after its inception in 1996: The interactive prover *Coq* was written in OCaml as of version 5.10 [FHB+97] in 1997. Damien Doligez, a principal developer of the OCaml language, is also one of the developers of the Zenon system in Table 3.1.

The *E*-column marks the provers which have built-in equality handling. This refers to specific equality rules within the implemented calculus. Technically all provers are capable of handling equations by adding equality axioms to a given problem; that way the equality symbol is treated as a normal predicate symbol, and the axioms express the semantics intended for equations (see Section 2.2). However, this method provides no means to control the generation of new terms. This can lead to an explosion of the search space, making axiomatization less effective than dedicated equality rules. On the other hand a prover with built-in equality handling is likely to be more complex than one without (note the KLOC-differences between Darwin and E-Darwin, or between iProver and iProver-Eq, in both cases the latter being an equality-enhanced version of the former), resulting in more implementation work and also in more computational overhead which can reduce the performance on problems without equations. The addition or omission of built-in equality handling is therefore a design choice to be considered with care.

[4] `http://www.ocaml.org`

The P and M columns indicate whether a prover is capable of providing a proof or a model respectively for the problems it solves. The ability to present proofs might be taken for granted at first when dealing with theorem provers. However, automatically derived proofs may consist of millions of inference steps, and the actual proofs may be too unwieldy to be of interest compared to a compact *yes/no* result regarding the satisfiability. This means a prover does not necessarily have to provide proofs in order to be useful within an application scenario. Nevertheless, most of the listed provers have the ability to present proofs, with the exception of Paradox, which is primarily a model generator. One could further distinguish between systems which present proofs in the form of a listing of the inference steps performed, and those provers which provide a condensed proof. The former is relatively simple to implement, as the prover can print these inference steps on the fly during the derivation process. A disadvantage is that such a trace is likely to contain steps which are formally not required to prove the result. The ability to provide a condensed proof consisting only of the essential inference steps is thus more desirable. Its implementation comes with its own difficulties, though: The necessity of an inference is only known in hindsight, so the condensed proof cannot be printed during the reasoning like a trace. Instead the prover must keep a log of the inferences it performs throughout the derivation process and then extract the essential steps once the problem has been solved. This extraction is non-trivial, and the accumulation of the log can require substantial system resources. For example, when a tracing prover backtracks during a derivation, it can free memory immediately by discarding all invalidated clauses, whereas a prover with condensed proofs must preserve a representation of some of these clauses in the inference log.

The ability to provide models is less common and mostly dependent on the calculus. Tableaux and instance-based methods have an advantage here in that their proof search is based on the construction of models. Geo is the only listed resolution-based system with the ability to generate models, its geometric resolution calculus is a heavily modified variant of classic resolution.

The final column (*NF*) shows whether the prover comes with its own normal form converter or not. As mentioned before, virtually all ATP systems perform their inferences on some clausal form representation of the original problem. Commonly this is CNF, but there are exceptions like Geo, which uses so-called *geometric formulae*. As logical problems are often expressed in full FOL syntax including quantifiers, implications and other symbols, they have to be converted into a form compatible with the calculus of the respective prover. This non-trivial task is referred to as *clausification*, and the implementation of a clausification algorithm is a *clausifier*. Not every prover includes its own built-in clausifier, so some have to rely on external solutions. Stand-alone clausifiers exist, for example *FLOTTER* [NRW98]. It is also possible for theorem provers to employ other theorem provers solely for their clausification features. A third group of provers in the table requires no clausification (marked as "n/a"), because these systems process only problems consisting of unit equations. The Muscadet system is unusual in that it operates on the full FOL syntax instead of on a clausal form; its earliest version was actually incapable of handling CNF problems. As the current version does not have this limitation, it counts as a prover with built-in NF-transformation for the purposes of this overview.

This listing has provided a glimpse of the plethora of attempts to tackle the undecidability of first-order logic. However, eventually any prover will be

confronted with problems it cannot solve in reasonable time or within reasonable memory limits. Reasoning within limits is thus a reality of ATP, and some systems are more successful at this than others, making it necessary to evaluate and compare automated theorem provers.

3.3 Theorem Prover Evaluation

Theorem provers were originally devised as a means to find solutions to mathematical problems. However, as their designs progressed and diversified, and as their implementation techniques and their underlying calculi grew more refined, forming the research subfield ATP, the need arose to collect and even create logical problems specifically for the purpose of evaluating and comparing theorem prover implementations. There are several reasons for this: Primarily a problem library allows benchmarking different provers against each other. Some provers may be better than others in general, or some provers may be especially suited to certain types of problems. The results of such comparisons enable the prospective user to make an informed choice regarding which ATP system to employ.

A problem library is also helpful for internal use during the development of a theorem prover, in particular when evaluating modifications to the system. An automated theorem prover must make its own decisions on how to handle a given problem. Modifications to the calculus or implementation optimizations may not be as beneficial as intended when applied to all problems in general, and a large scale evaluation can reveal the problem classes which suffer or profit from the prover changes. Modifications may also turn out to be a trade-off, for example allowing a prover to solve more problems in a shorter time while requiring more memory, which may be acceptable or not depending on the application scenario.

Finally, a problem library is an indispensable tool for debugging. Theorem provers evolve continuously and increase in complexity as they are optimized to increase their capabilities. This also increases the risk for incurring unsoundness of the system due to programming oversights. As machine-generated proofs can be both very unwieldy and have little significance within the context of an application, it is not unlikely that a prover's compact result regarding the satisfiability of a problem will be taken at face value, while the underlying proof is discarded, even though it might contain evidence of flawed reasoning. ATP systems must therefore be tested thoroughly in order to warrant this trust.

For first-order logic systems the largest benchmarking library of this type is the *TPTP* (Thousands of Problems for Theorem Provers) [SS98], developed and maintained by Geoff Sutcliffe and available for download.[5] This collection deserves a closer inspection due to its significance in the ATP community. Beginning in 1993, the TPTP has at the time of this writing reached the release version 5.3.0, and currently it contains 15,550 FOL problems. These have diverse origins, for example mathematical problems, logical puzzles, logically formalized tasks from real-world applications, and also test cases specifically tailored to expose common oversights in prover implementations. The problems are specified in TPTP-syntax, basically a machine readable notation of first-order logic and some extensions. Using this syntax the problems are defined in

[5] `http://www.tptp.org`

clause normal form (CNF) or as first-order formulas (FOF), the latter allowing quantifiers and logical operators. To illustrate this, we consider the following logical formula:

$$\forall x \forall y (p(x, y) \rightarrow \exists z (q(f(x, y), z)))$$

As it is a fully-fledged formula, the FOF-notation of the TPTP-syntax is used to obtain this expression:

```
fof(name, axiom, (![X,Y]:(p(X,Y) => (?[Z]:(q(f(X,Y),Z)))))).
```

The same formula has the following clause normal form (note the introduction of the Skolem-function g):

$$\{\neg p(x, y),\ q(f(x, y),\ g(x, y))\}$$

The TPTP-expression in CNF-notation for this is:

```
cnf(name, axiom, (~p(X,Y) | q(f(X,Y), g(X,Y)))).
```

A problem can have an arbitrary number of expressions. The TPTP problems range between having a single clause or formula and those consisting of millions. Most expressions tend to be axioms like in the examples above (indicated by the second argument). However, other expression types are possible, for example conjectural formulas, which a theorem prover must treat appropriately during a proof atempt.

As the problems are intended for prover testing, each problem is annotated by its correct result - as far as it is known, because a number of the problems have not yet been solved.[6] Each problem also has a difficulty rating between 0.0 and 1.0. This rating is established in cooperation with the ATP community. The TPTP-maintainers test the TPTP under standardized conditions using various theorem provers. Some of these have the *SOTAC* status (*state-of-the-art contributor*), meaning that each of them solves some hard problems that few other provers (preferably none) can solve. The SOTAC provers determine the difficulty rating: The more provers can solve a given problem, the lower its rating becomes. Hence, a problem with a rating of 0.0 has been solved by all SOTAC provers, and a 1.0-rated problem by none. As of TPTP v5.3.0 there are 3,405 FOL problems that cannot be solved by any of the SOTAC provers. The ratings can fluctuate between TPTP releases: As provers improve, more and more systems may be able to prove a problem. On the other hand it is also possible for provers to lose the ability to solve a problem, for example because some modification which offers an improvement to the overall performance happens to have an adverse effect on the processing of some particular problem. Therefore the ratings can also increase, and the TPTP may contain problems which cannot be solved by any current prover, yet for which correct results and proofs are known from earlier testing. Currently the testing is performed using 62 different theorem provers, not all of them SOTAC provers.[7] The most successful system is Vampire, which solves 53% of the whole TPTP library, whereas the average system solves 14%. These numbers illustrate that ATP testing can be

[6] This result annotation merely indicates whether a problem is a theorem, satisfiable, unsatisfiable etc. It does not provide a proof, which would be prohibitively large.

[7] Most recent result listing: `http://www.cs.miami.edu/~tptp/TPTP/Results.html`
Our data is based on the results retrieved on 1 May 2012.

an expensive endeavour: Granting a per-problem time limit of 300 seconds (the minimum used in the official TPTP tests), even the best provers may require several weeks for a full pass through the TPTP when using a single CPU core.

Nevertheless, constant testing is critical to ensure the soundness of ATP systems. In our work on the theorem provers E-KRHyper and E-Darwin we experienced that even minor changes could lead to unforeseen soundness errors. Sometimes performance improvements would also reveal old soundness bugs that had not become apparent in earlier tests, because the older prover version had been too slow to reach the breaking point within the time limit. Typically a soundness bug would manifest itself in less than 10 TPTP problems out of approximately 15,000. An early version of E-Darwin even passed a full TPTP test run with excellent results, despite a glaring soundness bug which was only discovered because the performance was suspiciously good. On one occasion Geoff Sutcliffe reported to us coming across a soundness bug in E-KRHyper on a single problem; it turned out that the error had existed since the earliest version of the prover (the non-equality KRHyper), but it had remained dormant because it would only lead to an incorrect result on a variant of that particular TPTP problem obtained by changing the clauses into a different order. Therefore more diversity in test problems is helpful for the ATP developer, and the TPTP remains open to the submission of new problems.

The TPTP and its syntax have found widespread acceptance in the ATP community. The TPTP-maintainers provide an online interface[8] to 64 theorem provers, only 11 of which provide no native ability to read the problems without conversion.

Closely linked to the TPTP is the annual ATP competition *CASC*[9] [PSS02, SS06], which has been held at the CADE[10] or IJCAR[11] conferences since 1996. The participating systems run on computers provided by the organizers and have to solve TPTP problems within a time limit, usually 240 seconds per problem. The problems are grouped into divisions according to problem features (for example problems in CNF or FOF, satisfiable or unsatisfiable problems, with or without equality, and so on), and participants can choose in which divisions to compete. All competition problems are randomized versions of the official benchmark problems (predicate and function symbols have been renamed, and the order of axioms and clauses has been changed), and the divisions also contain problems which have not yet been released in any version of the TPTP. These measures ensure that the participating provers cannot win by simply matching the competition problems against a database of known TPTP problems and their solutions. As of 2008 the CASC has included a special division for very large problems, the LTB category (Large Theory Batches). Here provers have to solve a series of problems which share a large axiom set. This setup offers opportunities for strategies that have no relevance in the regular divisions. For example, provers can save time by only reading the axioms once and reusing them for all problems, instead of starting anew for each problem. As of 2009 the CASC has begun branching out by including a category for HOL provers.

Other AR projects similar to the TPTP and the CASC exist, usually specializing in other logics or other types of reasoning. The *SATLIB* [HS00] was

[8] `http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP`

[9] CADE ATP System Competition: `http://www.cs.miami.edu/~tptp/CASC/`

[10] Conference on Automated Deduction: `http://www.cadeinc.org`

[11] International Joint Conference on Automated Reasoning: `http://www.ijcar.org`

founded in 1998 and is a problem collection for SAT solvers. This library has nearly 50,000 problems, all of which are generated variants of 97 base problems, though. Development of the SATLIB appears to have ceased in 2003. However, the annual SAT conference[12] continues to hold a competition for SAT solvers with a number of participants comparable to the CASC. The *SMT-LIB* [BST10], founded in 2003, is a benchmark library for SMT solvers. It contains almost 100,000 problems divided into 22 logics, the largest rivalling the TPTP with 14,335 problems. The SMT-LIB was inspired by the TPTP, and it has established a standardized problem and theory specification language. It also has a similar associated competition for SMT solvers, the *SMT-COMP*.[13] *QSTRLib*[14] is a collection of problems in qualitative spatial and temporal reasoning. The project is a recent development, at the time of this writing the library contains 126 problems.

[12] SAT - International Conference on Theory and Applications of Satisfiability Testing: `http://www.lri.fr/SAT2011`

[13] `http://www.smtcomp.org`

[14] `http://qstrlib.org`

Chapter 4

Question Answering

The second major research field forming the foundation of this dissertation is question answering (QA), which concerns itself with the development of methods for the automatic derivation of answers to questions phrased in a natural language. For example, given the question *"Which planet is closest to the Sun?"*, a QA system should respond with *"Mercury"*. QA aims at making vast quantities of knowledge easily accessible in an intuitive way, and it can therefore be regarded as the next step beyond conventional search engines. Research in QA is wide in scope and sets no particular conditions on how the answers are to be procured, though the mainstream approaches tend to combine aspects from natural language processing (NLP), information retrieval (IR) and knowledge representation (KR), as well as assorted AI techniques like machine learning (ML). One should distinguish between closed-domain and open-domain question answering. A closed-domain system can only answer questions regarding a specific, demarcated topic. This type of expert system can obtain its answers from a knowledge base constructed by knowledge engineers. Open-domain systems on the other hand try to answer questions about arbitrary topics. This means that extensive amounts of knowledge must be available to the QA system. At this scale a manually engineered, all-encompassing ontology is unfeasible. Instead the knowledge has to be drawn from existing textual sources like encyclopedias, processed in a manner that makes their contents accessible to a machine.

The goals of QA are ambitious: An ideal QA system would have a textual understanding comparable to that of a human being, yet operate at considerably higher speed and with a much larger and more reliable storage of knowledge. QA therefore represents an advanced application of AI, and many of its problems are not yet well understood. The development of QA systems is very much at an early stage, and none of the existing systems achieve a performance that would allow a ubiquitous usage similar to search engines.

The desire to make vast quantities of knowledge readily accessible can be traced back to antiquity, to the collection of books in the first libraries and then the further compilation of knowledge into encyclopedias, the first possibly being the *Naturalis Historiae* by Pliny the Elder in 77 CE. These still very common solutions offer knowledge in a locally concentrated form which shortens the search process for a particular piece of information, a process that nevertheless must be carried out by a human reader. This task has become more and more daunting with the ever increasing accumulation of knowledge, as the tra-

ditional storage methods have not scaled well. For example, the current edition of the *Encyclopædia Britannica* [Hoi10] consists of 32 volumes, with the index alone taking up two of the books, and one volume (the *Propædia*) serving as guidance for the rest by providing an outline of knowledge. Their considerable physical dimensions render these knowledge stores impractical for portable use or as quick reference tools. Manual searching is hampered by coarse indexing, and connections between related facts may not be apparent due to the linearity of books. The cost of such volumes prevents widespread distribution, and errors and obsolete knowledge cannot be easily corrected, necessitating frequent replacement of the entire encyclopedia.

A first attempt to improve upon this state was the *Repertoire Bibliographique Universel* [Ray94], designed by Paul Otlet and Henri La Fontaine in 1895. This database catalogued knowledge on index cards containing bibliographic references, dossiers and images. The cards and their contents were sorted according to a fine-grained decimal classification system which attempted to map all human knowledge. Over the following decades this collection was housed in various institutions in Brussels, and it grew to a size exceeding 15 million cards. Cross-references between the cards anticipated the hyperlinks of today. A commercial search service allowed remote users to send queries by mail or telegraph, with copies of cards being sent back as results. The project met its end in 1939 when most of the collection was destroyed during the German invasion of Belgium, so this database always remained confined to the pre-digital technology of its time. However, its organization pioneered new ways of storing knowledge, separating facts from the linear structure of books and linking them in a web of semantic references.

Going further, Otlet predicted more advanced methods for the future, culminating in a global network of screen-equipped workplaces that could be used to access a central electro-mechanical archive. His death in 1944 precluded him from seeing such visions become reality, and his work was largely forgotten, to be rediscovered only later when others had begun implementing comparable approaches. Thus it may have been Vannevar Bush who influenced the coming developments more directly with his seminal essay *As We May Think* [Bus45] in 1945. Here he identified the problems of the traditional techniques of knowledge storage, and he proposed the *memex* device which would store large quantities of data, augmented by hidden link annotations that would allow quick access to related facts within the database.[1] These speculations inspired Douglas Engelbart, who in 1962 proposed using computers to store knowledge in an associatively linked form resembling human memory and thinking [Eng62]. He also stressed the importance of natural language and vaguely referred to possible question answering capabilities of future systems. In 1968 Engelbart's research group demonstrated the implementation of many of the ideas in what is today sometimes called the "Mother of all Demos", as the innovations presented there have since become commonplace in human-computer interaction. This includes the linking of knowledge as hypertext, but not question answering - indeed, the accompanying conference paper specifically mentions the missing QA: *"A third type of service operation that will undoubtedly be of significant aid to studying is question answering. We do not have this type of service."* [EE68].

[1] Incidentally, among other technologies this essay also predicted theorem provers. Bush's proposed symbolic logic machine would have required turning a hand crank, though, so one could argue whether to call it an automated or an interactive theorem prover.

However, QA had already been in development in computational linguistics. The *Baseball* system [GWCL61] of 1961 could answer English questions about a database of baseball matches. In the *SHRDLU* project [Win71] of 1968 the computer reasoned about a simulated world of blocks, the aptly named *BLOCKS world*, and answered simple English questions about the state of the blocks. As mentioned in the introduction (Chapter 1), the QA system of Black [Bla68] in 1968 even used deduction. These and other systems of subsequent years operated on very small, closed domains, usually specified by handwritten ontologies. The difficulty of encoding large quantities of knowledge limited the practical usage of the early QA systems even within their closed domain.

This situation has started to change in recent years. Advances in computational linguistics have lead to improved parsers which offer better automatic processing of natural language texts. Ever larger collections of knowledge are available in digital form, and the progress in hardware development allows more complex operations on more data within still reasonable time limits. Instead of specifying ontologies by hand, it has become increasingly feasible to derive knowledge bases automatically by parsing and encoding text collections. Such collections have become easily available with the widespread adoption of computers and the internet, for example in the form of electronically published articles and books, text corpora and online sources like *Wikipedia*[2], *DBpedia*[3] [BLK+09] and *WolframAlpha*.[4] There are also ambitious ontology projects like *Cyc* [Len95], *SUMO* [NP01] and *YAGO* [SKW08] which offer extensive curated knowledge bases. Formally prepared information about natural languages is available in databases like *WordNet* [Fel98] and *GermaNet* [HF97]. The modern QA researcher can thus draw from significant resources and is no longer required to start from scratch when developing a QA system.

FALCON [HMM+00] from the year 2000 is an example of a QA system which combines existing resources like WordNet with various methods from NLP and information retrieval in order to achieve open-domain capabilities. A more recent example is *True Knowledge* [TP10], an English language open-domain QA system accessible on the web.[5] Its knowledge base consists of a proprietary core ontology augmented both by structured knowledge drawn from Wikipedia and by information supplied by its users.

QA gained prominence when the *Watson* system [FBCC+10] of IBM[6] successfully participated in the American television quiz show *Jeopardy!*[7] in February 2011, winning the game by defeating two human former champions. Watson is notable for its scale; the system employs numerous different strategies in parallel on a dedicated computer cluster equipped with nearly 3,000 processors. It may be the first open-domain QA system to achieve the answering speed and accuracy that would be expected from an alternative to search engines, although its prohibitive hardware requirements will likely prevent widespread usage for the time being.

[2] `http://www.wikipedia.org`
[3] `http://dbpedia.org`
[4] `http://www.wolframalpha.com`
[5] `http://www.trueknowledge.com`
[6] `http://www.ibm.com`
[7] `http://www.jeopardy.com`

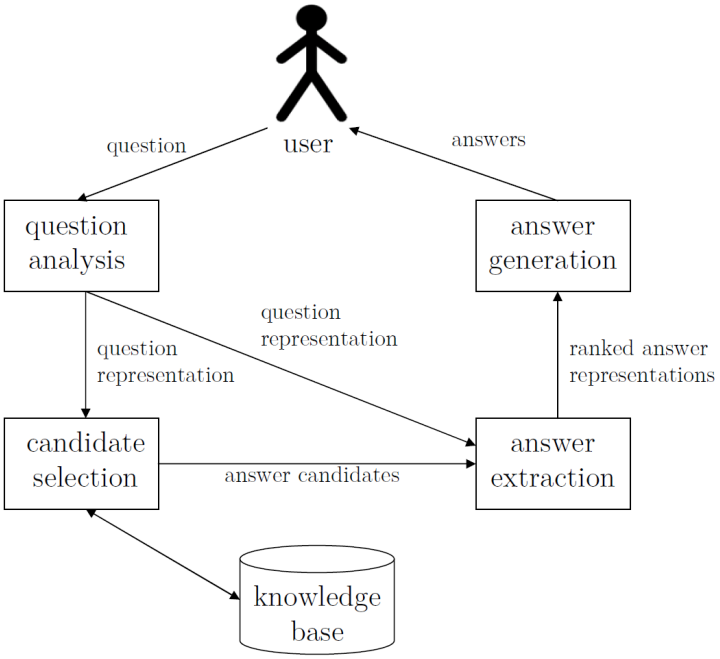


Figure 4.1: Generic core architecture for a QA system

2011 also saw the electronics company Apple[8] incorporating QA features[9] into its popular *iPhone* line of mobile phones. An internet connection is required, as most of the QA processing is done on Apple's servers. For the time being Apple still advertises this as an experimental "beta" software feature.

Research in QA is ongoing and garnering more interest as working systems are becoming more feasible. Languages other than English are considered as well, for example by the French *FIDJI* system [TM10], the Romanian *RACAI* system [ISC+09], and the Spanish *MIRACLE* [MGdPSPB+09], among many others. LogAnswer is a German language QA system.

4.1 QA System Implementation

The ambitious goals of question answering hold many challenges for QA system implementations, and there is great variety in how developers attempt to overcome these obstacles. Nevertheless it is possible and useful to identify a generic core architecture (see Figure 4.1, based on [HG01]), as it helps in obtaining an overview of the problems and their relations to each other. A fundamental component is the knowledge base, from which the answers will be extracted. The knowledge base may simply be a corpus of natural language documents, although for the sake of efficiency it is likely to be an easily searchable database of knowledge expressed in some machine-readable knowledge representation formalism. Accumulating the knowledge base is a major challenge, as its magnitude

[8] `http://www.apple.com`
[9] `http://www.apple.com/iphone/features/siri.html`

prohibits a manual formalization, requiring sophisticated parsers, lexicons and translators instead. Nevertheless it is an almost indispensable requirement for a useful QA system: In a good knowledge base the knowledge is separated from the original textual sources and compounded into essential facts, which are more easily accessible and which also allow a significant reduction of redundancy. For example, the two sentences

"China has a population of 1.3 billion."

and

"1.3 billion people live in China."

contain the same[10] knowledge, which could be expressed and condensed as a logical fact *population(china, 1.3billion)*, a notation that lends itself to database storage and queries. As mentioned in the introduction to this chapter, existing efforts at ontologies and knowledge databases can allow developers to save some work when building a QA system, but the integration of a third party knowledge source into a language and knowledge processing framework comes with its own challenges.

Apart from the knowledge base the generic QA system has four major components, which correspond to four phases of processing a question:

1. *Question analysis:* The natural language question entered by the user is analysed and turned into internal representations suitable for the subsequent processing. It is common to determine the type of the question - for example, it may be easily recognizable whether the question is asking for a person (*"Who... ?"*), a date (*"When... ?"*), or a location (*"Where... ?"*), and this can be important information during the search for an answer. Some syntactic and semantic information may be gleaned from the question, like whether it contains any proper names and what these refer to. This phase may also deal with spelling errors and ambiguities, and it can enter into a dialogue with the user in order to resolve such problems.

   Typical issues in this phase involve question parsing, simple semantic analysis and translation of the natural language question into a formal representation.

2. *Candidate selection:* A question representation compiled by the previous phase serves to retrieve the so-called candidates from the knowledge base. Depending on the system, a candidate may for example be a document, a sentence, a phrase, or a semantic network fragment, and it has some likelihood of containing an answer to the question. Candidates are found using fast, shallow information retrieval methods, and the candidate selection is thus a filtering that narrows down the vast knowledge base into manageable fragments which are small enough to be handled by more elaborate methods in the next phase.

   The major challenge here is the fast yet accurate retrieval of the candidates. This requires the identification of suitable recognition criteria, a knowledge base organization which facilitates finding candidates according

[10] Some might argue that the sentences are not equivalent due to differences in emphasis, but such subtleties are beyond the scope of QA for now.

to such criteria, and often the creation of an evaluation method that enables a ranking of candidates, allowing the dismissal of a weaker candidate in favour of a more promising one.

3. *Answer extraction:* Matching a question representation from the first phase (not necessarily identical to the one used for the candidate selection) against the candidates, deeper analysis methods attempt to find actual answers. Different candidates may yield multiple different answers, and the system may have varying confidence in these answers. Also, if the system uses a formal knowledge representation, then the answers may not be in a form suitable for a human user. Ranked by their confidence values, the answers are forwarded to the final phase.

   The typical problems in this phase concern the answer identification, as simple information retrieval is usually insufficient at this stage. Instead the semantics must be taken into account, while still remaining within the time constraints of the usage model. The answer ranking again requires an identification of criteria and a quick computation of these for the given answers.

4. *Answer generation:* The best answer or the best answers found during the answer extraction are translated into a natural language suitable for the user. Depending on the nature of the knowledge base this can simply mean handing over the respective candidate phrase (if the knowledge base is basically a corpus), or it may require a more involved translation from a formally represented answer into natural language.

   This phase is likely the least challenging, although the natural language answer formulation may be complex. A system may also be designed to enter a dialogue with the user at this point, in order to justify its answers or to adjust the answer presentation based on user feedback.

The generic architecture is a rough guide to QA systems, and actual systems may vary greatly in how the phases are implemented, or even use an entirely different architecture. It should be obvious, though, that QA system implementations are generally grander in scale than the tightly focussed automated theorem provers. QA systems consist of many components and subcomponents which may have different developers and which can be switched during the development of the system or even at runtime. Unlike the typical automated theorem prover, the common QA system is not easily portable, as its significant resource requirements tie it to a certain hardware infrastructure.

4.2 QA System Evaluation

It is difficult to conduct a systematic evaluation of QA systems. In principle a number of criteria for the usefulness of a real-world QA system have been identified [BCC+03]:

Accuracy: The answers must be correct, and giving no answer is preferable to giving a wrong answer.

Completeness: The answers must be complete regarding the questions. For example, given the question *"Who was Nikola Tesla?"*, the answer *"A*

man." would be accurate, yet hardly satisfying to the user. To achieve completeness a QA system may have to gather facts from different sources and merge them into an answer.

Relevance: The answers must be relevant within the context of the respective user. For example, a QA system should be able to correctly interpret the question *"What did the president talk about in his address to the nation?"* based on the user's location or nationality, interests, and data about current events.

Timeliness: The QA system must respond immediately, and it must include current knowledge.

Usability: The QA system must be able to obtain knowledge from any source, and it must provide answers in any format desired by the user. This includes textual, audio and video sources as answers.

At the current state of the art QA systems would score lowly in all categories, and indeed research is at such an early stage that accuracy and completeness still have to receive the lion's share of attention in order to achieve a system that is at least remotely useful. In practice an answer that is both accurate and complete is usually simply referred to as being *correct* or *right*, and any answer that is not correct is called *incorrect*, *wrong* or *false*. We adopt this usage in the sequel.

However, even when neglecting the other criteria in favour of accuracy and completeness it remains problematic to judge QA systems objectively, as the perception and acceptance of the answers may vary from user to user. For example, a user might ask an ambiguous question like *"How large is China?"*, and then reject an answer despite it being correct for a different interpretation (in this case, *"large"* could refer both to the area and to the population, among other things). A user might also disagree when there is no universal consensus regarding an answer and the QA system picks a position not matching the user's, for example when answering *"How many continents exist on Earth?"* (where some cultures consider Asia and Europe to be separate continents while others do not). The criterion of completeness is also difficult to assess, since one user might be satisfied with the comprehensiveness of an answer while another might want more detail. For example, when asking *"When did John F. Kennedy die?"*, some users would be content to learn the year, some would prefer the exact date, and someone interested in the criminal case might even want to know the precise time of day. For other questions (like *"What is a black hole?"*) a detailed scientific answer can be necessary to satisfy some users while being rejected as incomprehensible by laymen.

Large scale studies with numerous human participants can nevertheless provide insights into the performance of a QA system, but such experiments do not lend themselves to the everyday evaluation that is common in ATP development where automated testing is available. An effort to introduce automatic testing of QA systems is hampered not only by the aforementioned lack of objective criteria, but also by the fact that the systems so far are simply not very good at phrasing natural language answers, often taking them from the textual sources where they were found, for example when responding to the singular *"What is a dog?"* with the plural *"mammals related to wolves"*. A human user may easily

ignore such minor mismatches, but for an automatic evaluator this is more difficult. Paraphrasing is generally problematic, as any answers beyond the most simple factoids[11] can be phrased in different ways, which requires an automatic evaluator to have its own deep semantic understanding of answers.

For these reasons the evaluation of QA systems is still mostly manual work. However, despite these difficulties there are competitions for QA systems. The TREC[12] [Har95] provided a QA competition track mostly for English language systems from 1999 to 2007, and the CLEF[13] has been holding similar competitions since 2003, offering more choices of language. Since CLEF is the current major competition venue for QA and LogAnswer participated in CLEF for several years, it will be the focus of this dissertation when it comes to competitive evaluations of QA systems. As there is no obvious ideal evaluation method, the specifics of the competitions have varied greatly over the years. Generally though it can be said that the participating QA systems have to answer a number of questions referring to a set of documents. These documents are made available in advance, and they contain enough knowledge to answer the questions. The participants have ample time to preprocess the documents and to incorporate them into their respective knowledge bases. At the start of the actual competition the participants receive the questions, and then they have a limited amount of time to produce the answers. These answers are assessed by a panel of human judges. The evaluation criteria are usually limited to accuracy and completeness (subsumed under the notion of the correctness of an answer). The questions are phrased in such manner that relevance is not much of an issue, and the time limits are generous compared to real-world usage, often approximately 10 minutes per question. Answers are only accepted in textual form, so the criterion of usability is not applied.

The complexity of QA system design means a steep entry threshold for participants, so the number has usually been low - for example, 12 systems took part in the QA track of CLEF 2011 [PHF+11]. To accommodate systems for different languages, the base documents and questions are provided in translations. Also, as QA systems are not easily portable, the participants use their own hardware infrastructures and send the answers online. These factors impede the arrangement of equal conditions for all participants, and hence CLEF is less competitive in nature than ATP competitions like CASC, which features clear winners. Instead, CLEF is very much a competition for a field that is still in early development, and each year's subsequent analysis by the organizers tends to emphasize not only individual results, but also general research directions, promising approaches and overall results like the total number of questions answered correctly by any system. A more detailed account of various CLEF competitions, including the performance of LogAnswer, will be given in Chapter 14.

[11] We use the term *factoid* in the sense it has acquired in the QA community, referring to very short answers, often a single word or number. A factoid question is a question that can be answered by a factoid.

[12] Text Retrieval Conference: `http://trec.nist.gov`

[13] Cross-Language Evaluation Forum: `http://www.clef-campaign.org`

Chapter 5

Combining Question Answering and Automated Reasoning

At their core, QA systems and automated theorem provers have similar goals: Both attempt to glean some piece of information out of a set of data - QA searches a knowledge base for facts which answer a question, a prover searches a set of formulas for a subset that proves a theorem. However, their approaches are fundamentally different. A theorem prover uses a sound and preferably complete calculus, because automated reasoning is interested in irrefutable results to its investigations. Question answering on the other hand is willing to accept results that are "good enough", provided that they can be achieved within reasonable time and resource limits. Arguably this acceptance of imperfection is a necessity, borne out of the wish to create usable systems despite the many open challenges in QA research. However, this mindset has also led to developments regarding how to deal with the imperfections, developments that are valuable in their own right. In a modern QA system the components are finely tuned to each other in order to overcome their respective weaknesses and to attain a general robustness of the total system.

5.1 Advantages and Problems of Conventional QA Methods

QA methods usually aim for robustness, the ability to produce useful results despite flawed input data and imperfect analysis techniques. This robustness enables the rapid extraction of answers from vast knowledge bases, but the correctness of these answers is very much hit-or-miss - the average accuracy in the QA competition of CLEF 2008[1] was 23.6% [FPA+08] - because the robustness comes at the cost of depth. The methods employed in QA operate mostly on the

[1] As of CLEF 2009 the QA track has been exploring new directions and undergoing significant organizational changes, which means a great variation in the results in recent years. 2008 was the final year of the original competition approach, so we consider its results more representative of the state of the art than the experimental later tracks.

syntactic level of language, which prevents them from uncovering and utilizing deeper semantic relations in the search for an answer. This is sufficiently fast to extract, evaluate and rank hundreds of answer candidates for a given question, but the individual evaluations are semantically shallow: The approach effectively relies on sheer quantity to find a candidate that both contains an answer and is simple enough to be grasped by the limited "understanding" of the QA system.

A restriction to shallow methods may be acceptable during the candidate selection phase (see Section 4.1), which has to perform the preliminary filtering on the entire knowledge base. In the subsequent answer extraction it becomes a real hindrance, however, as it is easy to identify situations in which syntax alone is insufficient.

For a beginning, consider this pair of question Q and answer candidate C_1:

Q**:** *"Which mountain range did Hannibal cross?"*

C_1**:** *"Hannibal crossed the Alps."*

With the knowledge of C_1 it is possible to produce *"the Alps"* as an answer to Q. This is not too difficult even for shallow methods, as the sentence structures are simple and many words match between question and candidate. Nevertheless some semantic background knowledge would be helpful to validate the answer, because otherwise a sentence like *"Hannibal crossed the Rhône."* could also appear to contain an answer.

The purely syntactic approach can be derailed using synonyms:

Q**:** *"Which mountain range did Hannibal cross?"*

C_2**:** *"Hannibal overcame the Alps."*

For this a QA system has to resolve the synonymous usage of the verbs *"to cross"* and *"to overcome"*. Fortunately this is no major hurdle, and synonyms are even handled by some conventional search engines.

It turns more troublesome when paraphrases are used:

Q**:** *"Which mountain range did Hannibal cross?"*

C_3**:** *"The Carthaginian general led his army over the Alps."*

Here background knowledge about Hannibal is required to identify his reference in C_3. It is also necessary to recognize that *"x leads y over z"* implies *"x crosses z"*, at least in this particular case.

A different kind of difficulty results from information spread over multiple sentences.

Q**:** *"Which mountain range did Hannibal cross?"*

C_4**:** *"Hannibal led his army against Rome. They crossed the Alps in 218 BCE."*

While this simple example is likely compact enough to be managed by some shallow methods (a coreference resolution algorithm would have to handle the anaphoric *"they"*), the information required to answer a given question may be scattered throughout several sentences further apart than here, or even in multiple documents.

The examples above justify the involvement of semantics in question answering, but their solution does not require much in the way of reasoning, as in theory most cases could be resolved by shallow methods bolstered with an extensive lexicon that allows looking up words that are semantically equivalent to those in the question and the candidate. This is not sufficient when the candidate implies the answer in a less direct manner, for example:

Q: *"Which mountain range did Hannibal cross?"*

C_6: *"Hannibal invaded the Roman homeland from the North, marching his army all the way from Iberia to Gaul and then turning towards Italy."*

Here the answer *"the Alps"* does not occur in the candidate at all, but it could be deduced by a system that has access to geographical background knowledge that would allow it to map the path described in C_6 and discover which mountain range forms a natural barrier to be overcome. This example is extreme in the amount of background knowledge required, and it likely goes beyond the capabilities of most QA systems. But reasoning can be necessary for much simpler examples:

Q: *"How many moons does Mars have?"*

C: *"The moons of Mars are called Phobos and Deimos."*

Here the answer *"two"* can be derived by counting the names in C and considering that the usage of the verb *"to call"* ties each name to one entity.

Reasoning can be useful in dealing with ontological class hierarchies:

Q: *"What is the largest mammal?"*

C: *"The Blue Whale is the largest animal."*

In this case *"mammal"* must be recognized as a subclass of *"animal"* such that *"largest"* holds for both, resulting in the answer *"Blue Whale"*.

The examples given here are not meant to be an exhaustive listing of phenomena that can be handled by deduction, as this would require a focus on the mechanics of natural language that is outside the scope of this dissertation. Numerous other examples could be given where it is necessary to resolve or utilize spatial and temporal relations, concatenations of relations, conjunctive, disjunctive or negated phrasings, and more. Suffice it to say that shallow syntactic methods can quickly reach their limits, and at that point a firmer grasp of semantics is required to arrive at an answer. This leads us back to automated reasoning.

5.2 Advantages and Problems of AR Methods

By representing knowledge in a formal language like first-order logic and applying an automated theorem prover, the issues listed in the previous section can be addressed in order to achieve a deeper reasoning on the semantics. Suitably selected predicates and function symbols can represent attributes, actions and relations between concepts and entities. For example, the sentence

S_1: *"The Venus is a planet circling the Sun."*

could be translated into this formula:

F^{S_1}: $planet(venus) \wedge circles(venus, sun)$

Given that a knowledge base tries to separate the knowledge from the original text, it is not necessary to try to preserve the structure from the textual source by using one formula per sentence. Hence a knowledge base would rather store the representation of S_1 as separate FOL facts:

$F_1^{S_1}$: $planet(venus)$

$F_2^{S_1}$: $circles(venus, sun)$

Carefully chosen symbols then allow a theorem prover to draw conclusions based on the semantics. For example, let us assume that the knowledge base also contains a representation of the sentence

S_2: *"Planets circle around stars."*

in the form of:

F^{S_2}: $\forall x \forall y(planet(x) \wedge circles(y) \rightarrow star(y))$

Then a theorem prover would be able to deduce that the Sun is a star. Such deductions may involve many steps - the whole *raison d'ètre* of ATP systems is to find proofs that are too large and complex to be derived by human logicians, and preferably to do so in short time. Theorem provers are therefore optimized to handle large numbers of inferences and long derivation chains.

In addition to the reasoning strength of theorem provers, the usage of logic and AR offers further advantages, addressing the aforementioned weaknesses of conventional QA methods. Synonyms can be dealt with during the translation into FOL, where they are all mapped to one canonical form. For example, the two sentences

S_3: *"The Evening Star is a planet."*

S_4: *"The Morning Star is a planet."*

could each be translated into the FOL fact

F^{S_3,S_4}: $planet(venus)$

which would be subsumed by the knowledge base above.

FOL is a declarative language, which is why ordering of formulas should have no bearing on the derivation of a proof. As such the spatial proximity of facts within a set of formulas has no relevance for the calculus, and ideally neither for

the prover. Hypothetically the difficulty of shallow methods to utilize dispersed facts from different sources does not apply to automated reasoning, as all facts can be stored in one knowledge base, separated from the syntactic structure of the text. We will see that in practice a formal QA knowledge base can grow so large as to exceed the capabilities of ATP systems, which reintroduces the proximity problem as the prover has to focus its efforts on a more or less well-chosen fragment that may have omitted crucial knowledge found elsewhere in the knowledge base. However, the methods of automated reasoning are intended from the outset to handle declarative representations. Therefore they are at least at an advantage regarding dispersed information when compared to syntactic methods that cannot ignore the syntactic structure.

Unfortunately it is not a trivial proposition to use a theorem prover for question answering. Three problem areas can be identified: Representing the knowledge base in first-order logic, dealing with the size of the knowledge base, and overcoming the brittleness of deduction. These three areas will be examined in more detail.

5.2.1 Logical Knowledge Base Representation

The knowledge base of the QA system must be available in a first-order logic format for the theorem prover. The translation of natural language into FOL is problematic for several reasons. First of all, there is no single way of representing knowledge with the means of FOL. For example, the sentence

S_1**:** *"Betelgeuse is a star."*

can be translated into

$F_1^{S_1}$**:** $star(betelgeuse)$

or into

$F_2^{S_1}$**:** $is(betelgeuse, star)$

or into

$F_3^{S_1}$**:** $being\_action(a_1) \wedge present(a_1) \wedge subject(betelgeuse, a_1) \wedge object(a_1, star)$

among countless other variations, in increasing order of reification. The simple $F_1^{S_1}$ has the advantage of being compact, but it leads to a high number of predicate symbols when translating a large knowledge base. This is impractical when formulating reasoning rules. These are preferably kept as general as possible in order to minimize their number, and since FOL does not allow predicates as variable arguments, the applicable predicates must be stated explicitly, increasing the amount of rules in order to cover all variations. The approach also lacks expressivity, for example in how to express the tense, which leads to confusion when a sentence like

S_2**:** *"Betelgeuse will become a black hole."*

is added to the knowledge base in the form of this formula:

F^{S_2}**:** $black\_hole(betelgeuse)$

A workaround would be to have different variants of each predicate symbol for every combination of tense, modality and so on, but this would lead to an even more excessive amount of predicates.

The more complex $F_2^{S_1}$ and $F_3^{S_1}$ lessen these problems by moving more and more information into the atoms, away from the predicates. $F_3^{S_1}$ in particular, by treating an action as a constant, allows specifying the tense as a predicate. This approach can achieve a high level of expressivity while maintaining a low number of predicates, provided they are chosen carefully. There is a risk that oversights during the early stages of this ontology design leave some aspect uncovered, making later additions necessary that can have repercussions for the existing knowledge base. The automatic translation from natural language is also more difficult, as more intricacies have to be captured.

Even the most detailed translation faces the problem that pure first-order logic has difficulties in dealing with all aspects of natural language and human reasoning due to a lack of expressivity. Modality, default reasoning and arithmetics are just some examples for this. Certain intricacies of natural language can cause surprising problems: For example, it may appear intuitive to simply translate adjectives as predicates, yet a sentence like

S_3**:** *"John is a fast runner, but not a fast cyclist."*

should not result in this contradictory translation:

$F_1^{S_3}$**:** $fast(john)$

$F_2^{S_3}$**:** $runner(john)$

$F_3^{S_3}$**:** $\neg fast(john)$

$F_4^{S_3}$**:** $cyclist(john)$

Extensions to first-order logic can provide some solutions to expressivity problems, but few theorem provers are capable of processing more than pure FOL.

When using very detailed knowledge representations the amount of literals obtained from a sentence is notably higher than with the simpler translations. This increases the size of the knowledge base, which leads us to the next problem area.

5.2.2 Size of the Knowledge Base

Automated theorem provers were originally intended to solve mathematical problems, and the specifications of such problems tend to be compact. The TPTP aims at collecting a representative sample of problems of ATP usage, and Table 5.1 attempts to illustrate the size range with different measures. The sizes are generally not very large. Almost half of the problems have at most 50 clauses or formulas; indeed, the median amount is 52. Over 85% of the problems have no more than 1,000 clauses or formulas, and only a few outliers exceed 100,000 or even a million. Measuring by the number of atoms results in a similar distribution, with about 77% of the problems having no more than 1,000. The median number of atoms is 184. Generally, the average difficulty the problems pose to a theorem prover increases along with their size. The average difficulty rating of the FOL problems in the TPTP is 0.56, indicating

clauses/formulas per problem	problems	average rating
at most 10	3,109 (20.0%)	0.32
11 - 50	4,561 (29.33%)	0.49
51 - 100	2,139 (13.76%)	0.51
101 - 1,000	3,511 (22.58%)	0.70
1,001 - 10,000	1,141 (7.34%)	0.78
10,001 - 100,000	889 (5.72%)	0.87
100,001 - 1,000,000	145 (0.93%)	0.96
over 1,000,000	55 (0.35%)	0.96

atoms per problem	problems	average rating
at most 10	1,702 (10.95%)	0.34
11 - 50	2,708 (17.41%)	0.42
51 - 100	1,785 (11.48%)	0.46
101 - 1,000	5,713 (36.74%)	0.59
1,001 - 10,000	2,083 (13.4%)	0.67
10,001 - 100,000	991 (6.37%)	0.81
100,001 - 1,000,000	513 (3.3%)	0.96
over 1,000,000	55 (0.35%)	0.96

Table 5.1: Statistics about problem sizes in TPTP v5.3.0: Only first-order logic problems were considered. Both subtables quantify the sizes of typical ATP problems by partitioning the TPTP set according to different measures. Column explanations: *clauses/formulas per problem* (upper table) and *atoms per problem* (lower table) - the number of clauses, formulas or atoms of each problem in the respective group; *number of problems* - the number of problems in each of the partitions (percentage with respect to the total number in parentheses); *average rating* - the average TPTP difficulty rating of the problems in each partition, ranging from 0.0 for the easiest to 1.0 for the most difficult.

that the average problem can be solved by slightly less than half of the current state-of-the-art provers. The problems with more than 1,000 clauses or formulas have an average rating of 0.83, which means that on average, each of them can be solved by less than one fifth of the provers. The largest problems (beyond 1,000,000 clauses or formulas) are rarely solved, and only by specialized systems that employ incomplete selection algorithms (see Section 12.2.2).

On the one hand, such large problems are clearly atypical for ATP usage, and they are contained in the TPTP mostly as a challenge specifically due to their abnormal size. The TPTP usually also includes smaller sized versions of the same problems, allowing ATP developers to incrementally work their way up towards the full problems, in order to test how much their systems can handle. On the other hand, the largest problems are exactly those originating in knowledge representation, and they include FOL versions of ontologies like Cyc and SUMO. They are more typical for the problems that a QA system will face, yet they do not even contain the massive amounts of world knowledge required for true open-domain question answering. Hence it is clear that a theorem prover within a QA system will have to handle logic problems that exceed the size the ATP system is intended for by several orders of magnitude. Under

these circumstances a normal theorem prover cannot maintain a performance suitable for real-world usage, as its data structures and algorithms will not scale sufficiently well.

There are two major ways in which excessive problem size will adversely affect an ATP system. Firstly there are effects on the implementational level. At the minimum one must expect relatively benign problems of this type in the form of an approximately linear increase of processing time and memory usage, caused simply by creating, processing and storing millions instead of hundreds of clauses. In practice these increases will likely be more severe and have more complex reasons, like certain design choices that do not scale to larger amounts of data. For example, in the clause indexing trees of the original KRHyper the child nodes of any given node are stored in a list structure, and accesses and searches are handled by simple list iterating operations. This is a compact and very efficient implementation as long as these nodes rarely have more than ten children, as is the case when processing normally sized logic problems. The same approach breaks down when nodes have thousands of children, as traversing these enormous lists repeatedly results in a dramatic increase of processing time for what should be simple and common-place indexing operations. Hash tables are preferable to lists in these cases, although their overhead is less efficient when dealing with normal-sized problems, where each table often only has to store a few elements.

Even more critical are situations when large problems exceed hard implementation limits. For example, the SPASS system has a limit on the amount of logical symbols it can store at runtime, and it will refuse to process a reasoning problem that crosses this boundary. This is a graceful way of coping with such problems, but it is only possible when the hard limit is known, for example when it has been explicitly implemented as a design choice, like KRHyper's inability to process clauses with more than 100 different variables. However, systems may also have unknown hard limits which go unnoticed due to the rarity of very large reasoning problems, and an attempt to process such a problem will then cause the prover to crash. An example of this are certain list functions in OCaml (like `List.map`) which have undocumented limits that can cause them to terminate a program when dealing with long lists of several ten thousand elements. Limitations like this are not obvious in a prover, and they may be difficult to diagnose and circumvent when encountered.

The second area where excessive size of reasoning problems causes difficulties is the level of reasoning strategy. Theorem provers tend to work in a saturation-based fashion where they begin with the axioms and then derive clauses bottom-up until they can refute the negated conjecture. The lack of goal-orientation in this approach is no severe obstacle when dealing with normal-sized reasoning problems, as heuristics can aid in the search for a refutation and all axioms are likely to be relevant for the proof. The latter is not the case with reasoning problems based on open-domain question answering. Here the axioms form the knowledge base, and as the knowledge base aims at providing knowledge to answer any question, the axioms have to cover a wide range of facts, and only a fraction of them will be necessary to answer any given question. Using the full knowledge base with all its axioms in the bottom-up manner will therefore bog down the reasoning in irrelevant conclusions. While it is trivial to determine the relevance of an axiom once a proof has been found, the same may be impossible to do beforehand. Axiom selection methods are therefore often heuristics which

do not guarantee completeness. Nevertheless it is essential for a theorem prover to use some form of axiom selection in order to cope with the size of QA-oriented knowledge bases.

5.2.3 Brittleness of Precision

The vast knowledge bases of open-domain QA systems are bound to have numerous omissions and errors due to the way they are created. Such flaws are carried over into the FOL representation for a theorem prover. However, unlike natural language processing systems a theorem prover is not robust against flaws in the data. Indeed, the opposite is the case: ATP design places great value on the soundness of the calculus and the implementation, so that the system can deliver reliable proofs with millions of inference steps. Granting the prover some leeway when drawing conclusions would break the strict soundness standard, and the usefulness of the proofs would be diminished. In a QA situation on the other hand some latitude may be required to achieve robustness. For example, consider the following question and answer candidate:

Q**:** *"How high is the highest mountain?"*

C**:** *"The Mount Everest is the highest mountain in the world. It is a part of the Himalaya mountain range. It is 8,848m tall."*

For a human reader the question should be easy to answer given the knowledge in C. However, let us assume that the parser of a knowledge base creation system has trouble resolving the anaphoric *"it"* in the third sentence of C, since it can refer to *"Mount Everest"* and *"Himalaya mountain range"* (and technically even to *"mountain"*, *"world"* and the first *"it"* in the second sentence, which is also ambiguous). The system decides to err on the side of caution when creating a FOL representation and assigns a fresh constant to this pronoun occurrence, thus generating a new entity rather than establishing a potentially false association between existing entities. A simplified excerpt of the resulting FOL knowledge base could look like this:

F_1^C**:** $\mathit{mountain}(\mathit{mount\_everest})$

F_2^C**:** $\mathit{highest}(\mathit{mount\_everest})$

F_3^C**:** $\mathit{is\_part\_of}(\mathit{mount\_everest}, \mathit{himalaya})$

F_4^C**:** $\mathit{has\_height}(c_1, \mathit{8{,}848m})$

...

Likewise, the question Q can be represented in logic form:

F^Q**:** $\exists x \exists y(\mathit{mountain}(x) \wedge \mathit{highest}(x) \wedge \mathit{has\_height}(x, y))$

Ideally a proof would instantiate the question variable y with *8,848m*, but this proof is impossible, as the variable x cannot be unified both with *mount_everest* and c_1. Also note that the transformation already assumes a successful canonization of *"high"* and *"tall"* into the predicate *has_height*. If this were not the case, the mismatch of predicates would add another obstacle to the proof.

Such difficulties must be expected when processing FOL representations of knowledge bases derived from textual sources. As the precision of logic calculi renders ATP systems too brittle to handle the flawed data, a theorem prover within a QA system must feature robustness enhancements that allow a more "flexible" reasoning while still preserving a degree of control over the soundness.

5.3 Issues and Goals of the Combination

The previous sections listed advantages and drawbacks of both conventional shallow question answering methods and the automated reasoning approaches in theorem provers. An integration of AR into QA should provide benefits in the form of increased reasoning strength, allowing the QA system to obtain non-obvious answers that are only contained in the knowledge base in implicit form, and which are out of reach for the conventional shallow retrieval methods of QA. At the same time the brittle precision of automated deduction must be augmented by the robustness of QA. Thereby both AR and QA use their respective strengths to ameliorate their mutual weaknesses. The integration must address the following major areas:

- a first-order logic representation of the knowledge base and the questions,
- enabling the theorem prover to cope with the large size of the knowledge base via prover optimizations and axiom selection methods,
- achieving robustness of the logical processing against flaws in the FOL knowledge base,
- usage of ATP proofs for answer generation.

Furthermore, the usage model of the QA system must be taken into account when integrating and adapting the theorem prover. If the QA system is supposed to operate in a fashion similar to conventional search engines, then it should achieve comparable response times, leaving little time for deduction. On the other hand, a QA system employed as a scientific investigation tool, for example, can have more generous time restrictions, because its users may accept longer reponse times as a trade-off for higher quality answers. The LogAnswer project, which forms the background of this dissertation work, has involved the creation of a research prototype for QA system with an integrated theorem prover, intended for the development and exploration of deductive QA methods in different usage scenarios.

5.4 The LogAnswer Research Project

The LogAnswer project - full German title *"Logische Antwortfindung über semantisch strukturierten Wissensbasen"*[2] - is a research cooperation between the AGKI[3] of the Universität Koblenz-Landau, Germany and the IICS[4] of the Fern-

[2] *"Logical Question Answering on Semantically Structured Knowledge Bases"*: `http://www.loganswer.de`

[3] Arbeitsgruppe Künstliche Intelligenz (*Artificial Intelligence Research Group*): `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IFI/AGKI`

[4] Intelligent Information and Communication Systems: `http://pi7.fernuni-hagen.de`

	AGKI (Koblenz)	IICS (Hagen)
group/project leader	Ulrich Furbach	Hermann Helbig
researchers	Björn Pelzer	Tiansi Dong Ingo Glöckner
student assistants	Markus Bender Timo Eifler Sarah Grebing	Anna Kämpchen Julia Kramme

Table 5.2: Research personnel of the LogAnswer project

Universität in Hagen, Germany. The project is funded by the DFG[5] under the contracts FU 263/12-1, HE 2847/10-1, FU 263/12-2 and GL 682/1-2, running from 2007 to 2012. Table 5.2 gives an overview of the project personnel.

The IICS of Hermann Helbig is experienced in computational linguistics and knowledge engineering. Previous to LogAnswer the group developed the *MultiNet* (**Multi**layered Extended Semantic **Net**works) formalism [Hel06] for knowledge representation. The IICS also implemented a number of natural language processing tools and resources related to MultiNet. This includes the knowledge engineering tool *MWR* (**M**ultiNet **W**issens**r**epräsentation[6]) [Gnö00] which enables the knowledge engineer to build and maintain MultiNet knowledge bases, and which supports the development of MultiNet applications. MultiNet representations can be generated automatically with *WOCADI* (**Wo**rd **C**l**a**ss based **Di**sambiguation) [Har03], a parser and semantic interpreter for the German language. The WOCADI parser translates textual sources into MultiNet, utilizing the IICS-developed *HaGenLex* (**Ha**gen **Ge**rma**n** **Lex**icon) [HHO03] in the process. The latter is a semantic digital lexicon covering about 23,000 concepts and 220,000 proper nouns. Using such resources the IICS also already developed one conventional QA system: *InSicht* [Har04], features a knowledge base automatically derived from 2.5 million sentences; the system participated in several CLEF competitions. With this wealth of experience and assets the IICS took the responsibility for the natural language and knowledge engineering aspects of LogAnswer.

The AGKI of Ulrich Furbach covers a wide range of AI topics, but one focus is on automated reasoning. Noteworthy is the development of the hyper tableaux calculus and its implementation in the KRHyper series of automated theorem provers. Besides working on the pure basics of automated reasoning research, the AGKI has always emphasized the practical usage of deduction, and the KRHyper provers have been embedded in a number of applications, for example in e-learning and in a mobile phone information system. Hence in the LogAnswer project the AR aspects are the responsibility of the AGKI.

Apart from developing and improving the LogAnswer prototype, the research groups have also performed continuous evaluations of the QA system and its components, both internally and in competitions. The progress of the project and notable developments and results have been presented on numerous conferences and workshops and published in several journal articles and other papers.

[5] Deutsche Forschungsgemeinschaft (*German Research Foundation*): `http://www.dfg.de`
[6] *MultiNet Knowledge Representation*

Chapter 6

The Deductive Basis - Hyper Tableaux

As discussed in Section 4.1, the implementation of a question answering system faces many hurdles due to the diversity of tasks involved and the multitude of open research problems. Starting this kind of work from scratch is a daunting proposition. The LogAnswer QA system therefore consists of several subsystems, many of which have a history starting long before LogAnswer, originally being developed as stand-alone systems or as components of other applications. The focus of this dissertation is on the deductive component, the automated theorem prover E-KRHyper. The work on this prover within LogAnswer was my responsibility. This chapter will introduce the formal basis, consisting of the hyper tableaux calculus and its equational extension, the E-hyper tableaux calculus. The prover itself will then be the described in the next chapter.

6.1 The Hyper Tableaux Calculus

The hyper tableaux calculus was developed by Peter Baumgartner, Ulrich Furbach and Ilkka Niemelä at the Universität Koblenz-Landau in 1996 [BFN96]. It combines features of analytic tableaux with positive hyper resolution [Rob65b], resulting in an efficient model generation and proof procedure for first-order theories. The ability to generate models is an advantage of tableaux over more purely resolution-based calculi. Traditionally tableaux methods have a problem in how to deal with variables shared between branches, leading to the concept of *rigid variables* and the necessity to apply substitutions across several branches simultaneously. This is not the case with hyper tableaux, where such variables are avoided by the means of ground substitutions, as will be explained below. The main benefits of this approach are the ability to use strong redundancy criteria, as all variables within a branch are universally quantified, and the fact that an implementation only has to work on one branch at a time, thereby saving memory. The calculus also has the desirable feature of proof confluence, eliminating the need for backtracking on the calculus level. As mentioned before, the following calculus summary will provide an overview and only focus on those details relevant for its implementation. For a more thorough description of the calculus and its completeness proof, see [BFN96].

6.1.1 Trees and Tableaux

Some preliminary notions specific to tableaux are required for an understanding of the calculus description.

A *tree* is a pair $(\mathbf{N}, \mathbf{E})$ consisting of a set of nodes $\mathbf{N}$ and a set of edges $\mathbf{E}$. A *labeled tree over a set* M is a pair $(\mathbf{t}, \lambda)$, where $\mathbf{t}$ is a finite, ordered tree, and where λ is a labeling function assigning an element of M to each non-root node of $\mathbf{t}$. The *successor sequence* of a node $\mathbf{N}$ in an ordered tree $\mathbf{t}$ is the sequence of nodes with immediate predecessor $\mathbf{N}$, in the order given by $\mathbf{t}$. A *literal tree* is a labeled tree over the set of literals. Given a clause set $\mathcal{C}$, the *clausal tableau* $\mathbf{T}$ of $\mathcal{C}$ is a literal tree $(\mathbf{t}, \lambda)$, in which, for each successor sequence $\mathbf{N}_1, \ldots, \mathbf{N}_n$ in $\mathbf{t}$ labeled with the literals $K_1, \ldots, K_n$, there is a substitution σ and a clause $\{L_1, \ldots, L_n\} \in \mathcal{C}$ with $K_i = L_i\sigma$ for every $1 \leq i \leq n$. [LMG94, BFN96]

A *branch* $\mathbf{B}$ of a tableau $\mathbf{T}$ is a sequence $\mathbf{N}_0, \ldots, \mathbf{N}_n$ of nodes in $\mathbf{T}$, with $\mathbf{N}_0$ being the root node of $\mathbf{T}$, each $\mathbf{N}_i$ being the immediate predecessor of $\mathbf{N}_{i+1} (0 \leq i < n)$, and $\mathbf{N}_n$ being a leaf of $\mathbf{T}$. The notation $\mathbf{B} \cdot K$ represents the tableau obtained from attaching a node labeled with K to the leaf of $\mathbf{B}$, while $\mathbf{B} \cdot \mathbf{B}'$ represents the tableau obtained from concatenating $\mathbf{B}$ and the node sequence $\mathbf{B}'$. The set $\lambda(\mathbf{B}) = \{\lambda(\mathbf{N}_1), \ldots, \lambda(N_n)\}$ represents the *branch literals* of $\mathbf{B}$. For the sake of convenience, we usually identify a branch $\mathbf{B}$ with its literals $\lambda(\mathbf{B})$, and hence use the notation $L \in \mathbf{B}$ if $L \in \lambda(\mathbf{B})$. A branch containing a contradiction is called *closed*, otherwise it is called *open*. If all branches of a tableau $\mathbf{T}$ are closed, then $\mathbf{T}$ is closed, otherwise $\mathbf{T}$ is open. Branches and tableaux are always finite.

Finally, to formalize the semantics of branches, we introduce the following notations. Given a formula F, let $\forall F$ denote its universal closure, i.e. all free variables in F are universally quantified. If $\mathcal{A}$ is a set of atoms, then $\mathcal{A}^\forall$ denotes the unit clause set of $\mathcal{A}$, with $\mathcal{A}^\forall = \{\forall A | A \in \mathcal{A}\}$. Furthermore, let $[\![\mathcal{A}]\!]$ denote the minimal Herbrand model of $\mathcal{A}^\forall$. These notions are extended to literals and sets of literals. Thus a branch $\mathbf{B}$ is unsatisfiable if there is no model for $(\lambda(\mathbf{B}))^\forall$. Also, $[\![\lambda(\mathbf{B})]\!] \models F$ denotes the minimal Herbrand model of $(\lambda(\mathbf{B}))^\forall$ satisfying a formula F. As usual this notion is extended to clauses as well as to sets of formulas and clauses. Analogous to previous abbreviations, $[\![\mathbf{B}]\!]$ will usually serve as a shorthand notation for $[\![\lambda(\mathbf{B})]\!]$.

6.1.2 Hyper Tableaux

Let $\mathcal{C}$ be a finite clause set. *Hyper tableaux* for $\mathcal{C}$ are inductively defined as follows:

Initialization step: A one node literal tree is a hyper tableau for $\mathcal{C}$. Its single branch is labeled as open.

Hyper extension step: If

1. $\mathbf{B}$ is an open branch with leaf node $\mathbf{N}$ in the hyper tableau $\mathbf{T}$, and
2. $C = A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n (m, n \geq 0)$ is a clause from $\mathcal{C}$ (referred to as the *extending clause*, and
3. σ is a most general substitution such that $[\![\mathbf{B}]\!] \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$ (referred to as the *hyper condition*), and
4. π is a purifying substitution for $C\sigma$,

then the literal tree $\mathbf{T}'$ is a hyper tableau for $\mathcal{C}$, where $\mathbf{T}'$ is obtained from $\mathbf{T}$ by attaching $m + n$ child nodes $M_1, \ldots, M_m, N_1, \ldots, N_n$ to $\mathbf{B}$ with respective labels

$$A_1\sigma\pi, \ldots, A_m\sigma\pi, B_1\sigma\pi, \ldots, B_n\sigma\pi$$

and labeling every new branch $(\mathbf{B} \cdot M_1), \ldots, (\mathbf{B} \cdot M_m)$ with positive leaf as open and every new branch $(\mathbf{B} \cdot N_1), \ldots, (\mathbf{B} \cdot N_n)$ with negative leaf as closed.

The substitution σ being "most general" in this context means that whenever $[\![\mathbf{B}]\!] \models \forall(B_1 \wedge \ldots \wedge B_n)\delta$ for some substitution δ, then $\sigma \leq \delta\,[vars(B_1 \wedge \ldots \wedge B_n)]$. The existence of a most general substitution for a given clause C and a branch can be decided by a finite number ($|C|$) of SLD resolution steps.

The application of the purifying substitution π is called *purification*. If a clause $A \vee B$ is pure, then $\forall(A \vee B) \equiv (\forall A \vee \forall B)$. Thus the purification process ensures the soundness of the calculus when splitting a hyper tableau branch into several branches by using an extending clause with multiple head literals.

6.1.3 Redundancy and Model Generation

A common problem in any type of reasoning is the size of the search space, which can grow significantly even when dealing only with Horn clauses, and more so when non-Horn induced branching is necessary. While the issue of excessive derivations is not quite as pronounced as long as equality is not involved, even a calculus primarily intended for logic problems without equations, such as the hyper tableaux calculus, is served well by taking measures against unnecessary inferences.

Additionally this becomes particularly urgent when considering that the calculus does not prescribe any order in which to carry out the hyper tableaux inferences, since it aims for proof confluence. As a consequence there is also no prohibition against making the same inference more than once, possibly becoming hung up in a cycle. Fortunately this is easily avoided using the notion of *regularity*:

Definition 6.1 (Regularity)**.** *A hyper tableau* $\mathbf{T}$ *is* regular *if none of its branches contains two nodes* N_1 *and* N_2 *with* $\lambda(N_1) = \lambda(N_2)$.

Ensuring regularity is a common method in tableaux calculi, and it justifies avoiding certain inferences, including repetitions of earlier extensions. However, in the hyper tableaux calculus an even stronger pruning method is possible, based on a criterion of *redundancy*, and this will lead directly to the model generation capability of the calculus.

Definition 6.2 (Redundancy in Hyper Tableaux)**.** *A clause C is redundant with respect to a set of atoms $\mathcal{A}$ if $[\![\mathcal{A}]\!] \models C'$ for all ground instances C' of C.*

A redundancy test may be applied to the label of each new node that is about to extend a tableau in a hyper extension step. For example, this criterion allows rejecting inferred literals that are subsumed by earlier branch literals, and hence redundancy subsumes regularity. Let $\mathbf{B}$ be an open branch in the hyper tableau $\mathbf{T}$ for the clause set $\mathcal{C}$. If every clause from $\mathcal{C}$ is redundant in $\mathbf{B}$, then $\mathbf{B}$ is *finished*, and the branch literals form a model for $\mathcal{C}$ as per Section 6.1.1, i.e. $[\![\mathbf{B}]\!] \models \mathcal{C}$. By computing multiple finished branches the hyper tableaux calculus can be used to enumerate models.

6.1.4 Hyper Tableaux Derivations

If a hyper tableau $\mathbf{T}'$ can be obtained from a hyper tableau $\mathbf{T}$ by applying clause C in a hyper extension step to branch $\mathbf{B}$ in $\mathbf{T}$ with a most general substitution σ and a purifying substitution π, then this will be written as $\mathbf{T} \vdash_{\mathbf{B},C,\sigma,\pi} \mathbf{T}'$. Let $\mathcal{C}$ be a finite clause set called the *input clauses*, then a possibly infinite sequence $\mathbf{T}_1, \ldots, \mathbf{T}_n, \ldots$ of hyper tableaux for $\mathcal{C}$ is called a *hyper tableaux derivation from* $\mathcal{C}$ if $\mathbf{T_1}$ is obtained by an initialization step, and $\mathbf{T_{i-1}} \vdash_{\mathbf{B_{i-1}},C_{i-1},\sigma_{i-1},\pi_{i-1}} \mathbf{T_i}$ for $i > 1$, some clause C_{i-1} and some substitutions σ_{i-1} and π_{i-1}. A hyper tableaux derivation which contains a closed tableau is called a *hyper tableaux refutation*. The hyper tableaux calculus is sound and complete [BFN96], a hyper tableaux refutation for an input clause set $\mathcal{C}$ is derived if and only if $\mathcal{C}$ is unsatisfiable. If a hyper tableaux derivation for $\mathcal{C}$ contains a tableau with a finished branch $\mathbf{B}$, then $\lambda(\mathbf{B})$ represents a model for $\mathcal{C}$, and hence $\mathcal{C}$ is satisfiable.

6.1.5 Hyper Tableaux Derivation Example

Figure 6.1 shows the final hyper tableau in a derivation for the clause set $\mathcal{C}$ consisting of the following three clauses:

(1) $p(a) \leftarrow$

(2) $q(x, y), r(y, b) \leftarrow p(x)$

(3) $\leftarrow q(x, x)$

Starting from the root node, the unit clause (1) is used to extend the initial branch in a trivial hyper extension step that requires an empty substitution σ. Then clause (2) is used in a hyper extension step with $\sigma = \{x \leftarrow a\}$, the most general substitution such that $[\![p(a)]\!] \models \forall(p(x))\sigma$. This causes a split which requires a purifying substitution $\pi = \{y \leftarrow a\}$ to avoid the variable y being shared between branches. From left to right, the first split branch is the one extended by $\neg p(a)$, the substituted body literal of (2). This contradicts the earlier node labeled with $p(a)$, and the branch is closed immediately. The second split branch has $q(a, a)$ as its split node label, the purified first head literal of clause (2). This branch can be extended with the negative unit clause (3), a step resulting in a single new node labeled with $\neg q(a, a)$, and which is therefore closed. The final split branch caused by the earlier hyper extension with clause (2) has $r(a, b)$ as its split node label. Here another hyper extension using (1) and (2) is possible, with the difference that this time the purifying substitution

$\pi = \{y \leftarrow b\}$ is chosen instead. This results in three split branches, the first one with $\neg p(a)$ being closed immediately as above. The other two branches remain open, since they contain neither any contradiction, nor is it possible to extend them any further in a non-redundant way - a third hyper extension with (1) and (2) would only produce new literals if a new purifying substitution was applied, but the previous two applications have already made use of the entire Herbrand universe $H = \{a, b\}$. Both branches are therefore finished, and each of them represents a model for the input clause set, i.e. $\{p(a), r(a,b), q(a,b)\} \models \mathcal{C}$ and $\{p(a), r(a,b), r(b,b)\} \models \mathcal{C}$ respectively.

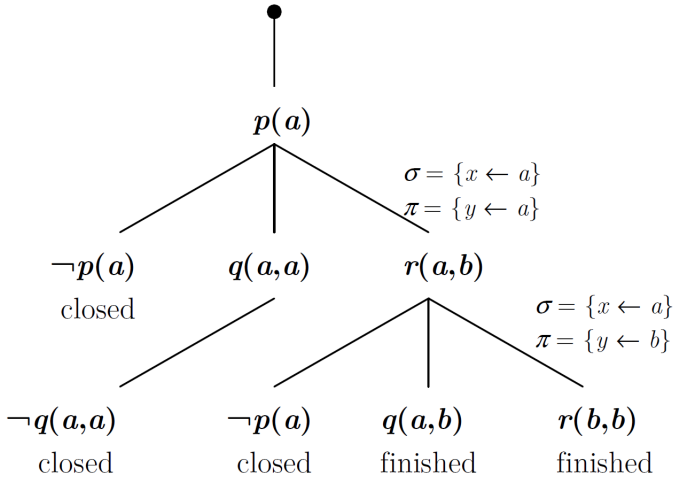


Figure 6.1: Example: hyper tableau for input clauses (1), (2) and (3)

6.2 The E-Hyper Tableaux Calculus

The hyper tableaux calculus has no native handling of equality, so theories with equations can only be treated by adding an axiomatization of equality (see Section 2.2). This solution is often highly inefficient due to the large number of generated axioms. The E-hyper tableaux calculus remedies this shortcoming by adding equality reasoning to hyper tableaux, using new inference rules that are in part adapted from the superposition calculus. The E-hyper tableaux calculus was developed by Peter Baumgartner, Ulrich Furbach and myself [1]. This development preceded LogAnswer and the work on this dissertation, but it is required for an understanding of E-KRHyper. The following summary of E-hyper tableaux will therefore focus on the essentials, similar to the foregoing description of the hyper tableaux calculus. For an in-depth exposition with proofs we refer to our aforementioned publication, or to its more comprehensive follow-up [2].

The calculi for hyper tableaux and E-hyper tableaux share many general ideas, but the changes are nevertheless significant. The addition of equality to the reasoning process allows replacing semantically equal terms occurring in clauses with each other, effectively creating new clauses. This is in contrast to hyper tableaux, where only new literals (or unit clauses) can be derived. A

major problem to overcome here is that the number of generated clauses can grow excessively (and indeed infinitely), in particular when a naïve handling of equality is utilized. The presence of equations within reasoning problems thus further compounds the issues discussed in Section 6.1.3, so there is a pressing need to curtail the clause count. Obviously, an ideal method would only generate those clauses that are necessary for the derivation of a proof. In practice this ideal is out of reach, but there are indeed methods to limit the inference results. A core concept used for this purpose in equational reasoning is that of a term ordering (see Section 2.2.3), which serves to restrict the creation of new clauses to those cases where the result is "simpler", loosely speaking, or at least not more complex than the original clause. But not only can new clauses be prevented, the creation of a new clause may also make existing clauses unnecessary. Redundancy criteria are defined based on the term ordering, allowing the identification and elimination of dispensable clauses. This is another major difference to the original hyper tableaux calculus, where anything that has been derived persists for the remainder of the derivation.

In contrast to the hyper tableaux calculus with its single inference rule, the E-hyper tableaux calculus makes use of four rules. Three of these deal with equations, hence taken as a group they will be referred to as the *equality rules*; they have been adapted from the superposition calculus. Each equality rule takes a set of clauses as input and in turn derives a new clause, provided certain conditions are met. The fourth so-called *split rule* serves to create a branch split from a disjunctive clause head. These four inference rules operate on clauses; they will then be lifted to tableaux in corresponding extension rules.

Further on two additional rules will be introduced which can remove unnecessary clauses from the tableau. These rules are optional; unlike the four inference rules they are not required for the soundness and completeness of the calculus, although their usage is recommended for the sake of efficiency.

As mentioned in Section 2.2.1, when describing the equational E-hyper tableaux calculus in the following, all atoms are assumed to have equational form.

6.2.1 Inference Rules

The sup-left rule (*superposition left*) selects a positive unit equation and applies it to a body literal of another clause.

$$\mathsf{sup\text{-}left}(\sigma)\quad \frac{\mathcal{A} \leftarrow s[l'] \simeq t, \mathcal{B} \qquad l \simeq r \leftarrow}{(\mathcal{A} \leftarrow s[r] \simeq t, \mathcal{B})\sigma} \quad \text{if} \begin{cases} l' \text{ is not a variable,} \\ \sigma \text{ is a mgu of } l \text{ and } l', \\ l\sigma \not\preceq r\sigma, \text{ and} \\ s\sigma \not\preceq t\sigma \end{cases}$$

If the sup-left rule is applied with clause C as left premise, D as right premise, the mgu σ and the conclusion E, then this inference instance will be denoted by $C, D \Rightarrow_{\mathsf{sup\text{-}left}(\sigma)} E$.

The unit-sup-right rule (*unit superposition right*) applies a positive unit equation to another positive unit equation.

$$\text{unit-sup-right}(\sigma) \quad \frac{s[l'] \simeq t \leftarrow \qquad l \simeq r \leftarrow}{(s[r] \simeq t \leftarrow)\sigma} \quad \text{if} \begin{cases} l' \text{ is not a variable,} \\ \sigma \text{ is a mgu of } l \text{ and } l', \\ (s \simeq t)\sigma \not\preceq (l \simeq r)\sigma, \\ l\sigma \not\preceq r\sigma, \text{ and} \\ s\sigma \not\preceq t\sigma \end{cases}$$

If the unit-sup-right rule is applied with clause C as left premise, D as right premise, the mgu σ and the conclusion E, then this inference instance will be denoted by $C, D \Rightarrow_{\text{unit-sup-right}(\sigma)} E$.

The third equality rule is the ref rule (*reflexivity*), which is an inference on a single clause. The ref rule removes a body literal whose both sides can be unified.

$$\text{ref}(\sigma) \quad \frac{\mathcal{A} \leftarrow s \simeq t, \mathcal{B}}{(\mathcal{A} \leftarrow \mathcal{B})\sigma} \quad \text{if } \sigma \text{ is a mgu of } s \text{ and } t$$

If the ref rule is applied to clause C with the mgu σ and the conclusion E, then this inference instance will be denoted by $C \Rightarrow_{\text{ref}(\sigma)} E$.

The split rule takes a clause with an empty body and, after the application of a purifying substitution, it creates a new unit clause for each head literal.

$$\text{split}(\pi) \quad \frac{A_1, \ldots, A_m \leftarrow}{A_1\pi \leftarrow \quad \ldots \quad A_m\pi \leftarrow} \quad \text{if} \quad \begin{cases} m \geq 2, \text{ and} \\ \pi \text{ is a purifying substitution for} \\ A_1, \ldots, A_m \leftarrow \end{cases}$$

If the split rule is applied to clause C with the purifying substitution π and the conclusions $A_1 \leftarrow, \ldots, A_m \leftarrow$, then this inference instance will be denoted by $C \Rightarrow_{\text{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$.

6.2.2 E-Hyper Tableaux

The aforementioned inference rules derive clauses from clauses. These rules will now be integrated into extension rules which apply the inferences in order to extend an E-hyper tableau. However, the tableaux used in the original hyper tableaux calculus are based on literal trees, where each non-root node is labeled with a unit clause. This is sufficient for the original calculus, as the set of non-unit clauses remains fixed throughout the derivation process; there is no inference method which creates a new non-unit clause. For the new equality rules this no longer holds true. A hyper tableaux calculus with equality must be able to store non-unit clauses in its data structures.

Thus, an *E-hyper tableau* $\mathbf{T}$ *over a signature* Σ is a labeled tree over the set of Σ-clauses. The notions introduced in 6.1.1 are extended to E-hyper tableaux appropriately: A branch $\mathbf{B}$ in $\mathbf{T}$ is a sequence of nodes $\mathbf{N}_1, \ldots, \mathbf{N}_n$ $(n \geq 0)$, and $\lambda(\mathbf{B}) = \{\lambda(\mathbf{N}_1), \ldots, \lambda(\mathbf{N}_n)\}$ is the multiset of clauses in $\mathbf{B}$. The shorthand $C \in \mathbf{B}$ will be used for $C \in \lambda(\mathbf{B})$. The notation $\mathbf{B} \cdot C$ represents the tableau obtained from attaching a node labeled with C to the leaf of $\mathbf{B}$, while $\mathbf{B} \cdot \mathbf{B}'$ represents the tableau obtained from concatenating $\mathbf{B}$ and the node sequence $\mathbf{B}'$. An initial E-hyper tableau $\mathbf{T}_0$ for a finite clause set $\mathcal{C}$ with n clauses $C_1, \ldots, C_n$ $(n \geq 0)$ consists of a single branch $\mathbf{B}$ of length n with $\lambda(\mathbf{B}) = \mathcal{C}$.

6.2.3 Extension Rules

If $\mathbf{T}$ is an E-hyper tableau with branch $\mathbf{B}$, then $\mathbf{T}$ can be extended by application of the following extension rules. The Equality-extension consolidates the three equality inferences:

$$\textsf{Equality}\quad \frac{\mathbf{B}}{\mathbf{B}\cdot E}\quad \text{if}\begin{cases} \text{there is a clause } C\in\mathbf{B},\\ \text{a fresh variant } D \text{ of a positive unit clause in } \mathbf{B}, \text{ and}\\ \text{a substitution } \sigma \text{ such that}\\ \quad C, D \Rightarrow_{R(\sigma)} E \text{ with } R\in\{\textsf{sup-left}, \textsf{unit-sup-right}\} \text{ or}\\ \quad C \Rightarrow_{\textsf{ref}(\sigma)} E, \text{ and}\\ \mathbf{B} \text{ contains no variant of } E \end{cases}$$

The Split-extension applies the split rule to a branch:

$$\textsf{Split}\quad \frac{\mathbf{B}}{\mathbf{B}\cdot A_1 \leftarrow^d \quad \ldots \quad \mathbf{B}\cdot A_m \leftarrow^d}\quad \text{if}\begin{cases} \text{there is a clause } C\in\mathbf{B}, \text{ and}\\ \text{a substitution } \pi \text{ such that}\\ \quad C \Rightarrow_{\textsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow,\\ \quad \text{and } \mathbf{B} \text{ contains no variant of}\\ \quad A_i \leftarrow \text{ for any } i = 1,\ldots,m \end{cases}$$

The annotation ^{d} marks the derived clauses as *decision clauses*.

6.2.4 Redundant and Subsumed Clauses

As mentioned above, a major problem to overcome in equality treatment is to keep down the number of generated clauses. For this it is necessary to define criteria by which clauses can be identified as not useful for the reasoning process. The first criterion is that of *redundancy*, a notion already introduced in Section 6.1.3, but which here requires a definition specific to E-hyper tableaux:

Definition 6.3 (Redundancy in E-Hyper Tableaux)**.** *Let D be a ground clause and $\mathcal{C}$ a set of clauses. Then $\mathcal{C}'$ denotes the set of all ground instances of all clauses in $\mathcal{C}$. The subset of clauses smaller than D in $\mathcal{C}$ is identified by $\mathcal{C}_D = \{C \in \mathcal{C}' \mid D \succ C\}$. A ground clause D is* redundant with respect to a clause set $\mathcal{C}$ *if $\mathcal{C}_D \models D$. If D is a non-ground clause then D is redundant with respect to $\mathcal{C}$ if every ground instance of D is redundant with respect to $\mathcal{C}$.*

Technically the criterion of redundancy can offer a justification for the elimination of numerous clauses within a derivation. However, in practice this is difficult to exploit to its full potential, since testing a clause for redundancy essentially requires proving it to be a theorem of some set of clauses - which is exactly the generally undecidable task that theorem provers struggle with in the first place. Therefore redundancy is often only determined and utilized in certain favourable circumstances, like the premise clause of some inference being made redundant by the conclusion clause. A general redundancy check for all derived clauses is not feasible, and thus redundancy will often remain undetected.

Another criterion for the irrelevance of a clause that is often simpler to test is that of *non-proper subsumption*.

Definition 6.4 (Non-Proper Subsumption)**.** *A clause C* non-properly subsumes *a clause D if there is a substitution σ such that $C\sigma = D$, and conversely, D* is non-properly subsumed by *C.*

In the description of the E-hyper tableaux calculus there will be an explicit distinction between redundancy as defined for E-hyper tableaux and non-proper subsumption. In a wider sense, when discussing different calculi and theorem provers, the general notion of redundancy and its elimination covers all such criteria and methods that deal with detecting and then simplifying or removing unnecessary clauses.

The criteria introduced in this section are utilized by he following rules which can change or remove clauses that have already been added to the tableau.

6.2.5 Deletion and Simplification Rules

Unlike the hyper tableaux calculus, the calculus explicitly provides methods that destructively modify the E-hyper tableaux in order to remove unwanted clauses. The Del rule (*deletion*) eliminates redundant or non-properly subsumed clauses (or more specifically, it overwrites such a clause with a trivially true unit clause).

$$\mathsf{Del}\quad \frac{\mathbf{B}\cdot C^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2}{\mathbf{B}\cdot\mathbf{t}\simeq\mathbf{t}^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2}\quad \text{if}\begin{cases}(1)\ C \text{ is redundant with respect to } \mathbf{B}\cdot\mathbf{B}_1,\\ \text{or some clause in } \mathbf{B}\cdot\mathbf{B}_1 \text{ non-properly}\\ \text{subsumes } C, \text{ and}\\ (2)\ \mathbf{B}_1 \text{ does not contain a decision clause.}\end{cases}$$

The notation $^{(d)}$ indicates that if the clause in the premise is a decision clause, then the resulting overwritten clause remains a decision clause.

The Simp rule (*simplification*) overwrites a clause with one that is smaller according to the term ordering.

$$\mathsf{Simp}\quad \frac{\mathbf{B}\cdot C^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2}{\mathbf{B}\cdot D^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2}\quad \text{if}\begin{cases}\mathbf{B}\cdot C\cdot\mathbf{B}_1\models_E D,\\ C \text{ is redundant with respect to } \mathbf{B}\cdot D\cdot\mathbf{B}_1,\\ \text{and } \mathbf{B}_1 \text{ does not contain a decision clause.}\end{cases}$$

In both rules, the scope of clauses that may subsume the premise clause C or make it redundant is limited to those above C in the branch $\mathbf{B}$ and those below until the first decision clause. This preserves the soundness of the calculus. A decision clause occurs after a branch split. If there is a decision clause below C, then C is a member of at least two branches resulting from concatenation to $\mathbf{B}$ in a Split extension. Only those clauses occurring above the first decision clause below C are guaranteed to be members of all branches resulting from splits below C, and if any of these clauses make C redundant (or subsume C non-properly), then C must be redundant (or non-properly subsumed) in all lower split branches. On the other hand, if a clause exclusively belonging to a lower split branch were used to overwrite C, then C would have been destructively modified or removed from all the other lower split branches as well, even though C may not have been redundant (or non-properly subsumed) in any of these.

6.2.6 E-Hyper Tableaux Derivations

A branch in an E-hyper tableau $\mathbf{T}$ is *closed* if it contains the empty clause $\square$. A branch is *open* if it is not closed. An E-hyper tableau is closed if all of its branches are closed, and it is open if at least one of its branches is open.

An *E-hyper tableaux derivation* of a clause set $\mathcal{C} = \{C_1, \ldots, C_n\}$ of Σ-clauses is a possibly infinite sequence of tableaux $\mathbf{D} = (\mathbf{T}_i)_{0 \leq i < \kappa}$ such that

1. $\mathbf{T_0}$ is the clausal tableau over Σ that consists of a single branch of length n labeled by the clauses $C_1, \ldots, C_n$, and

2. for all $i > 0$, $\mathbf{T}_{i+1}$ is obtained from $\mathbf{T}_i$ by a single application of the Equality, Split, Del or Simp rule to an open branch in $\mathbf{T}_i$.

A finite E-hyper tableaux derivation which contains a closed tableau is called an *E-hyper tableaux refutation*. The E-hyper tableaux calculus is sound and complete [1, 2], an E-hyper tableaux refutation for an input clause set $\mathcal{C}$ is derived if and only if $\mathcal{C}$ is unsatisfiable.

If an E-hyper tableaux derivation for $\mathcal{C}$ contains a tableau with an open branch $\mathbf{B}$, then a model may be obtained from this branch. However, here this is more complex than the model generation in the original hyper tableaux calculus, so it warrants its own section.

6.2.7 Model Generation

Let $\mathbf{B}$ be an open branch in the E-hyper tableau $\mathbf{T}$ for the clause set $\mathcal{C}$. If the conclusion of every inference applied to the clauses of $\mathbf{B}$ contains at least one clause that is redundant with respect to $\mathbf{B}$ or non-properly subsumed by clauses in $\mathbf{B}$, then $\mathbf{B}$ is *exhausted*. In that case the set of positive unit clauses in $\lambda(\mathbf{B})$ represents a model for $\mathcal{C}$, and $\mathcal{C}$ is satisfiable.

Note that compared to the full calculus description in [1], this definition of an exhausted branch is a severely simplified version which pertains to the implementation only.

A matter of further explanation is the choice of a purifying substitution π in the Split rule. So far no requirements have been posited apart from π being a ground substitution. If the set of function symbols in the signature of a given clause set contains at least one non-constant symbol, then the respective Herbrand universe is infinite. This means that there will often be an infinite number of purifying substitutions available, which is obviously a problem regarding the size of the search space, and especially when attempting to exhaust a branch. Therefore the eligible purifying substitutions will be limited to those that are irreducible. A substitution σ is *reducible* with respect to a clause set $\mathcal{C}$ if there is a term $t \in ran(\sigma)$, a unit clause $l \simeq r \leftarrow\in \mathcal{C}$ and a substitution γ such that $t[l\gamma]$ and $l\gamma \succ r\gamma$. Accordingly, a substitution is *irreducible* if it is not reducible.

6.2.8 E-Hyper Tableaux Derivation Examples

Figure 6.2 shows an E-hyper tableau that has been derived from the input clauses (1)-(6), which form the initial branch segment. The right branch of the tableau is open, and no further extension steps can be applied. Clause (14) cannot be split, as the resulting decision clause $r(b) \simeq t \leftarrow$ is already an element of the branch. Del steps can be used to further overwrite clauses (7) (redundant with respect to clause (8)) and (13) (redundant with respect to clause (14)).

A different example is found in Figure 6.3 which illustrates the usage of the Del and Simp rules. Starting with the input clauses (1)-(6), Equality extensions are used to construct the tableau consisting of clauses (1)-(14). Clause (10) is

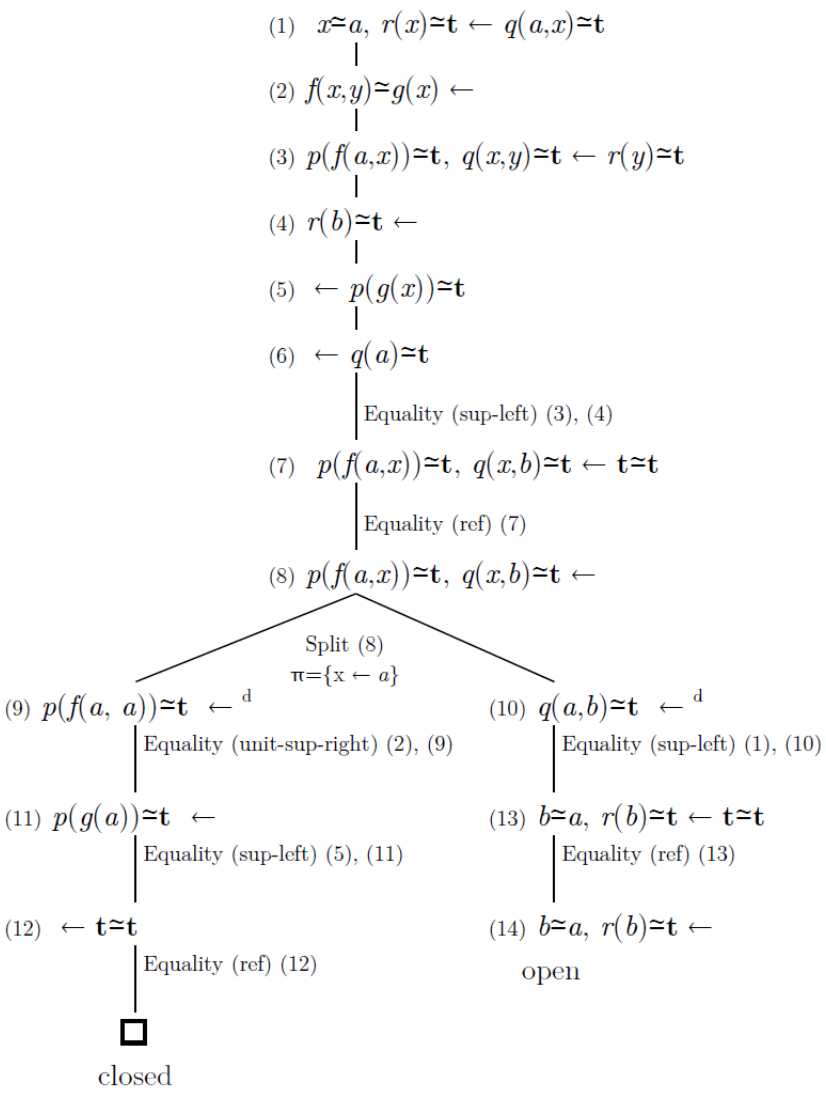


Figure 6.2: Example: E-hyper tableau for input clauses (1) - (6)

non-properly subsumed by clause (14) and can thus be deleted, replacing it with (10') $= t \simeq t \leftarrow$. Clause (9) is redundant with respect to clause (12) and the simpler clause (3). The Simp rule replaces clause (9) with (9') $= q(a) \simeq t \leftarrow$. As (9') itself is non-properly subsumed by (3), it can be deleted in a further step. Note that clause (1) cannot be deleted by clause (14), because there is a decision clause between the two clauses in the branch.

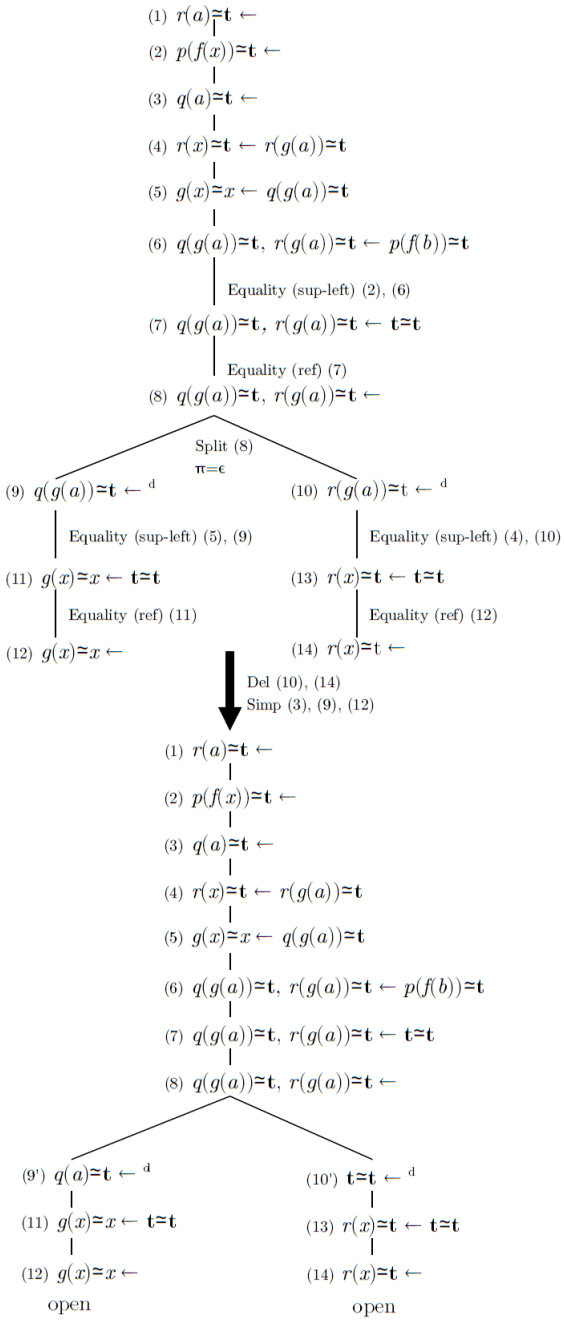


Figure 6.3: Example: E-hyper tableau for input clauses (1) - (6)

6.2.9 Hyper Extension in E-Hyper Tableaux

The E-hyper tableaux calculus takes its name from the original hyper tableaux it is based on. While it adds an efficient handling of equality, a critical observer might argue that it has lost the "hyper property" of the original calculus. In hyper tableaux the hyper extension inference (see Section 6.1.2) unifies all negative literals of the selected clause with branch labels, thereby emulating hyper resolution in its manner of processing multiple clause literals in a single inference step. Obviously this can be advantageous, as it allows deriving some crucial conclusion in less steps. For example, consider the clause set $\mathcal{C}$ which contains at least these clauses:

$$\begin{array}{ll} U_1: & q_1 \leftarrow \\ \vdots & \\ U_n: & q_n \leftarrow \\ C: & p \leftarrow q_1, \ldots, q_n \\ D: & \leftarrow p \end{array}$$

The units $U_1, \ldots, U_n$ can be added to the tableau immediately, and the resulting branch can be used in a hyper extension with the clause C to derive p, which leads to a refutation with the clause D. The hyper extension on C combines n unification operations into one inference. An altered calculus without the "hyper property" might break this down into separate steps, but that would require the storage of intermediate results. Worse, as the set $\mathcal{C}$ may contain further clauses and since calculi generally do not prescribe in which order to apply inferences, a derivation might become sidetracked by other inference possibilities after a few intermediate results.

Fortunately this is not a limitation for E-hyper tableaux. When specifying a calculus there is a motivation to restrict the number of different inference rules to the essentials, in order to simplify the proofs of soundness and completeness, and also as a matter of elegance. During actual usage of E-hyper tableaux a rule like the hyper extension could be very practical, and indeed this original rule can still be used, because it can be regarded as a shortcut that combines multiple applications of the E-hyper tableaux rules defined in Section 6.2.1.

Proposition 6.1. *The hyper extension step from the hyper tableaux calculus is applicable to E-hyper tableaux, as it is subsumed by the inference rules of the E-hyper tableaux calculus.*

Proof. Let $\mathbf{T} = (\mathbf{t}, \lambda)$ be an E-hyper tableau for a clause set $\mathcal{C}$ with an open branch $\mathbf{B}$ with leaf node $\mathbf{N}$. Let $\mathbf{U} \subseteq \lambda(\mathbf{B})$ be the set of positive unit clauses labeling $\mathbf{B}$. Let $C \in \mathcal{C}$ be a clause with $C = A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$ $(m, n \geq 0)$. Let σ be a most general unifier such that $[\![U]\!] \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$ (and hence $[\![\mathbf{B}]\!] \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$), analogous to the hyper condition (see Section 6.1.2), and let π be a purifying substitution for $A_1\sigma, \ldots, A_m\sigma$. We show that with the inference rules of the E-hyper tableaux calculus it is possible to derive the tableau $\mathbf{T}'$ which differs from $\mathbf{T}$ only in that branch $\mathbf{B}$ is appended by the branch segment $\mathbf{B}'$ which contains no decision clause, and that the split rule is applied to $\mathbf{B} \cdot \mathbf{B}'$ (when $m \geq 2$) with the purifying substitution π and the conclusions $A_1\sigma\pi \leftarrow, \ldots, A_m\sigma\pi \leftarrow$. These conclusions are equivalent to the nodes resulting from an application of the hyper extension step to a hyper tableau branch labeled with $\mathbf{U}$ and the extending clause C.

Since $[\![\mathbf{U}]\!] \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$, the branch $\mathbf{B}$ must contain a positive unit clause U_i of the form $L_i \leftarrow$ with $B_i\sigma \gtrsim L_i$ for each $1 \leq i \leq n$ (although it may be the case that $U_k = U_l$ for distinct body literals B_k, B_l of C with $k \neq l$). This allows for a sup-left inference with C as left and U_i as right premise and an mgu σ_i, for each B_i in C, with B'_i being the rewritten atom of that body literal in the resulting clause. Starting with the first body literal, there is then the inference $C, U_1 \Rightarrow_{\mathsf{sup\text{-}left}(\sigma_1)} C_1 = A_1\sigma_1, \ldots, A_m\sigma_1 \leftarrow B'_1\sigma_1, B_2\sigma_1, \ldots, B_n\sigma_1$. C_1 is attached to the leaf of $\mathbf{B}$ by the Equality-extension rule. Recall that $B_i\sigma \gtrsim L_i$ for each $1 \leq i \leq n$. This means that B_i and L_i unify, and therefore the rewritten B'_i has the form of a trivial equation $\mathbf{t} \simeq \mathbf{t}$. The resulting clause C_1 is thus available for a ref inference on its literal $B'_1\sigma_1$, using the empty mgu ϵ, and resulting in the clause $C'_1 = A_1\sigma_1\epsilon, \ldots, A_m\sigma_1\epsilon \leftarrow B_2\sigma_1\epsilon, \ldots, B_n\sigma_1\epsilon$. This clause C'_1 is again eligible for a sup-left inference and so on, meaning that a sequence of alternating sup-left and ref inferences can be carried out, each double inference of the form:

$$
\begin{aligned}
C_i \quad &= \quad A_1\sigma_1\epsilon \ldots \sigma_i\epsilon, \ldots, A_m\sigma_1\epsilon \ldots \sigma_i\epsilon \leftarrow \\
&\qquad\quad B_{i+1}\sigma_1\epsilon \ldots \sigma_i\epsilon, \ldots, B_n\sigma_1\epsilon \ldots \sigma_i\epsilon \\
D_i & \\
\Rightarrow_{\mathsf{sup\text{-}left}(\sigma_{i+1})} & \\
C_{i+1} \quad &= \quad A_1\sigma_1\epsilon \ldots \sigma_i\epsilon\sigma_{i+1}, \ldots, A_m\sigma_1\epsilon \ldots \sigma_i\epsilon\sigma_{i+1} \leftarrow \\
&\qquad\quad B'_{i+1}\sigma_1\epsilon \ldots \sigma_i\epsilon\sigma_{i+1}, \ldots, B_n\sigma_1\epsilon \ldots \sigma_i\epsilon\sigma_{i+1} \\
\Rightarrow_{\mathsf{ref}(\epsilon)} & \\
C'_{i+1} \quad &= \quad A_1\sigma_1\epsilon \ldots \sigma_{i+1}\epsilon, \ldots, A_m\sigma_1\epsilon \ldots \sigma_{i+1}\epsilon \leftarrow \\
&\qquad\quad B_{i+2}\sigma_1\epsilon \ldots \sigma_{i+1}\epsilon, \ldots, B_n\sigma_1\epsilon \ldots \sigma_{i+1}\epsilon
\end{aligned}
$$

Each conclusion clause is attached to the node created in the previous step of the sequence, finally resulting in the branch $\mathbf{B} \cdot \mathbf{B}'$, with the leaf node labeled with the final clause

$$C'_n = A_1\sigma_1\epsilon \ldots \sigma_n\epsilon, \ldots, A_m\sigma_1\epsilon \ldots \sigma_n\epsilon \leftarrow$$

Since the most general substitution σ allowing $[\![\mathbf{U}]\!] \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$ exists by definition, the most general unifiers σ_i (and the trivially empty ϵ) must exist as well for all steps of the inference sequence, and $\sigma = \sigma_1\epsilon \ldots \sigma_n\epsilon$. The last clause C'_n can therefore also be written as

$$C'_n = A_1\sigma, \ldots, A_m\sigma \leftarrow$$

We now compare this result to a hyper extension step performed on C and a branch that includes $\mathbf{U}$.

If $m = 0$, then C'_n is the empty clause ($C'_n = \square$), and $\mathbf{B} \cdot \mathbf{B}'$ is closed. The corresponding hyper extension would only add the negative leaves based on the body literals, and these resulting branches are closed immediately, thereby closing the branch that includes $\mathbf{U}$.

If $m = 1$, then C'_n is a unit clause. The corresponding hyper extension would add a single positively labeled node ($A_1\sigma$). No purification is required, hence π is empty, and the negative leaves are closed immediately, so they are of no concern for this proof. The single positive leaf is equivalent to the leaf of $\mathbf{B} \cdot \mathbf{B}'$.

If $m \geq 2$, then the corresponding hyper extension step uses the purifying substitution π to append m positive leaves labeled with $A_1\sigma\pi, \ldots, A_1\sigma\pi$, while the negative leaf branches again are closed immediately, so they can be disregarded. In this case a split inference is applicable to C'_n with the same purifying substitution π. The Split-extension rule attaches m nodes to the leaf of $\mathbf{B} \cdot \mathbf{B}'$, labeled with the positive unit clauses $A_1\sigma\pi \leftarrow, \ldots, A_m\sigma\pi \leftarrow$. Again, these nodes are obviously equivalent to the nodes derived by the hyper extension.

In summary, for every node derived by hyper extension on a clause C and a set of literals/unit clauses $\mathbf{U}$ an equivalent node can be derived by the means of the E-hyper tableaux calculus, and if the hyper extension closes a branch, then so do the rules of the E-hyper tableaux calculus. The hyper extension step is therefore subsumed by the E-hyper tableaux calculus, and it can be used on E-hyper tableaux without deriving any results that would be unavailable using only the inference rules of the E-hyper tableaux calculus. □

The above justifies the use of a hyper extension-like step within an implementation of the E-hyper tableaux calculus. This adapted hyper extension may not summarily replace its constituent inference steps, as the intermediate results may be necessary for the soundness and completeness of the calculus. However, it can serve as a shortcut, for example when a heuristic inference selection has estimated that the result can lead to a quick refutation. It is also possible that the result of this shortcut inference makes its selected clause redundant, and in that case the intermediate results are redundant as well, and the constituent inferences no longer have to be carried out one by one. As such this adapted hyper extension is an optimization with regard to an efficient operation, and theorem provers often contain many such shortcuts beyond the basic calculus implementation.

Chapter 7

The Theorem Prover E-KRHyper

This chapter will introduce the automated theorem prover E-KRHyper, the implementation of the calculi described in Chapter 6. E-KRHyper forms the deduction component of LogAnswer, and the work on this prover has been the main area of my research. Later chapters will detail the embedding of E-KRHyper into the QA framework of LogAnswer and the specific adaptations that became necessary, whereas the scope of this initial system description remains confined to the prover itself and its operation on first-order logic reasoning problems in general.

7.1 Background and Development History

E-KRHyper [16] (Knowledge Representation Hyper Tableaux with Equality) is a theorem proving and model generation system for first-order logic with equality. It implements both the hyper tableaux calculus (see Section 6.1) and the E-hyper tableaux calculus (see Section 6.2). E-KRHyper is geared towards embedding in knowledge representation applications, and it features an assortment of logic extensions to support such usage, for example stratified negation as failure, arithmetic evaluation and Prolog-like operators for lists and sets.

The prover is built upon the code of *KRHyper* [Wer03], which was developed at the Universität Koblenz-Landau as a first-order logic theorem prover and model generator based on the original hyper tableau calculus. After the creation of this calculus KRHyper passed through several prototype generations written in Prolog. This language offered the initial advantage of built-in unification and indexing operations. At the same time these operations were not as efficient as tailor-made solutions would have been, and this became increasingly important regarding an integration of KRHyper into real-world applications. This has led to the current version, written in the functional programming language *OCaml*.[1] Inofficially referred to as *KRHyper3*, the system was first released in 2003. It was implemented and subsequently maintained by Christoph Wernhard. KRHyper has been used as an embedded knowledge processing engine in several

[1] Objective Caml: `http://www.ocaml.org`

applications including content composition for e-learning [BFGHS04, BF05], document management [BFGH+03], database schema processing [BFGHK04], semantic information retrieval [BB04], ontology reasoning [BS06] and planning [BM04]. An excerpt of KRHyper has been ported to *Java ME*[2] and is employed for user profile matching on mobile devices [SK05].

I began development of E-KRHyper as part of my diploma thesis [14] and based it on KRHyper version 5.4.6, which is the most current version at the time of this writing, and which likely must be seen as the final version of the original KRHyper. Originally intended as an experimental fork of KRHyper for the testing of the E-hyper tableaux calculus, E-KRHyper has gone beyond the initial introduction of the equality rules and remained in continuous development since 2007. By now it subsumes and extends the functionality of its parent system while remaining backwards compatible and offering equal or better performance, including the correction of some flaws. Apart from its embedding in LogAnswer, E-KRHyper is used in the controlled natural language processor *PENG Light* [Sch09], and it is part of the *HETS*[3] framework for formal methods integration and proof management.

7.2 Usage Information

E-KRHyper is available under the GNU General Public License at the E-KRHyper website.[4] Its compilation requires an installation of OCaml 3.09.3 or a more recent version. Some supporting scripts coming with E-KRHyper distribution are written in Prolog and thus require a Prolog-installation, but none of them are necessary for the basic operation of E-KRHyper. The prover is intended for installation under POSIX-compliant operating systems like Linux and Mac OS and an installation within the Unix-like Cygwin[5] environment on Windows-based systems is supported as well. E-KRHyper has also been compiled on Windows without Cygwin, but this is experimental and unsupported.

For reasons of backwards compatibility to the original KRHyper, E-KRHyper can read input files in the Prolog-like *PROTEIN* syntax [BFK94], but unlike its predecessor it can do so natively in both prefix and infix notation, without requiring external input conversion scripts. E-KRHyper also accepts input in the TPTP syntax for first-order logic, both in the CNF-notation for clause normal form input and in the FOF-notation for formula expressions. A built-in clausifier (see Section 7.4.4) automatically converts formula expressions into the internally used clausal representation. This function can also be used without any proof derivation, allowing E-KRHyper to operate as a clausifier for other provers. The system can handle the `include`-command of the TPTP syntax by which shared axiom sets from the TPTP can be loaded automatically. E-KRHyper also supports loading multiple files with different notational standards.

Derivation results can be printed in the native tableaux-oriented syntax or in the TPTP and CASC-compliant *SZS result status* notation. In the case of a refutation E-KRHyper can print a proof for the closed tableau, and in the

[2] `http://www.oracle.com/technetwork/java/javame`

[3] Heterogenous Tool Set:
`http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets`

[4] `http://userpages.uni-koblenz.de/~bpelzer/ekrhyper`

[5] `http://cygwin.com`

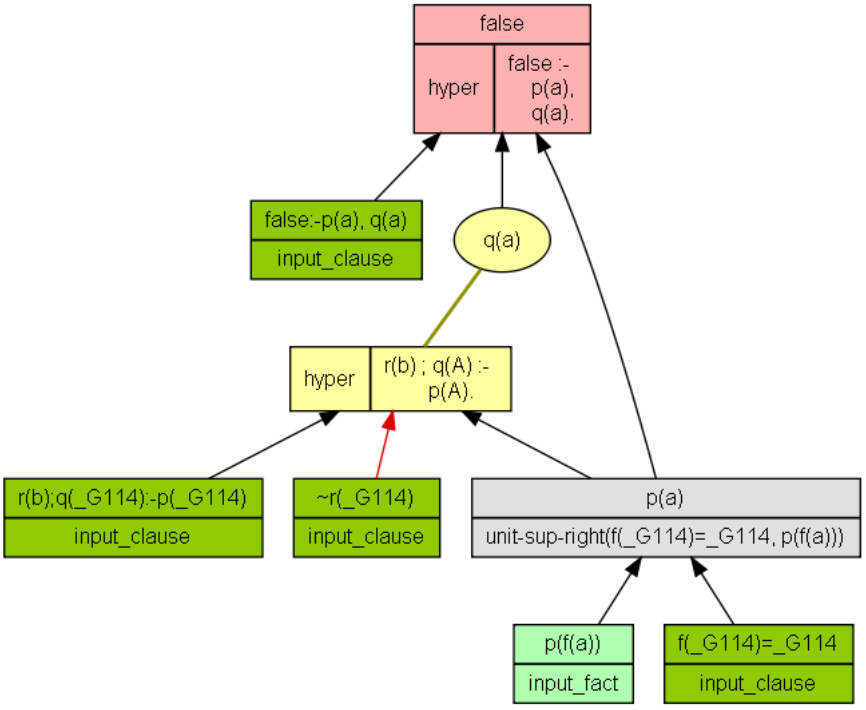


Figure 7.1: Proof visualization with E-KRHyper and Graphviz

case of an exhausted branch it can print the model for that branch, and it can print proofs for the literals of that model. The system can enumerate models, i.e. continue with other branches once one model has been found. Proofs are printed in a Prolog-compatible tree representation which is primarily intended to be machine-readable. These proofs contain only the relevant inference steps. The E-KRHyper distribution includes support scripts that can convert proof output into a human-readable inference sequence or into a graph specification that can be visualized using Graphviz.[6] Figure 7.1 shows such a graphical proof representation for the refutation of this clause set:

(1) $\leftarrow p(a), q(a)$

(2) $\leftarrow r(x)$

(3) $q(x), r(b) \leftarrow p(x)$

(4) $p(f(a)) \leftarrow$

(5) $f(x) \simeq x \leftarrow$

As E-KRHyper is geared towards embedding, it features an interactive mode in which it can communicate with applications or the user over STDIN and STDOUT or other user-specified channels. Numerous flags and parameters allow the user to control the proof derivation, including the specification of memory and time limits.

[6] `http://www.graphviz.org`

7.3 Proof Procedure

A calculus usually does not prescribe the order in which its inference rules should be applied, and for proof-confluent calculi like the hyper tableaux and the E-hyper tableaux calculus this order should not matter regarding soundness and completeness. However, an implementation that is intended as an effective tool must take matters of efficiency and fairness into account in choosing when to apply which inference rules to which premises. An unfair strategy can miss even simple refutations. Consider this example clause set:

(1) $p(f(x)) \leftarrow p(x)$

(2) $p(a) \leftarrow$

(3) $\leftarrow p(f(a))$

Clause (1) can be applied in a hyper extension step to the literal created by the unit (2), resulting in a new tableau literal $p(f(a))$. Now (1) can be applied again to that new literal, and then to the result of that inference, and so on. This cycle can go on indefinitely, while a single application of (3) to $p(f(a))$ could have closed the tableau.

Actually, this example already assumes some consideration for efficiency. An even cruder strategy could keep applying clause (1) to $p(a)$ over and over again. A regularity check would recognize that the result is already found in the branch and thus prevent it from extending the tableau indefinitely, but this check would not prevent the repeated futile attempts at applying (1) to $p(a)$.

Many design choices in the original KRHyper were inspired by deductive databases, both due to the intended usage of the system and to the background of its developer. The overarching strategy is based on *semi-naive evaluation* [Ull88], a technique that seeks to minimize the repetition of inferences with the same premises while ensuring fairness. The pseudo-code Algorithm 7.1 shows the basic semi-naive evaluation strategy, slightly adapted to the terminology of theorem provers.

Algorithm 7.1 The *semi-naive evaluation* strategy has its roots in deductive databases, and it is shown here using ATP terminology.

```
facts := unit clauses from input reasoning problem;
rules := input reasoning problem \ facts;
conclusions := all inferences with rules and facts;
while (conclusions ≠ {}) ∧ (contradiction not found) do
    newFacts := conclusions;
    conclusions := all inferences with newFacts,
                   using other premises from rules ∪ facts;
    facts := facts ∪ newFacts;
end while
if contradiction found then
    return contradiction
else
    return fixed-point
end if
```

The algorithm maintains two principal sets, the *facts* and the *rules*, with the former being the unit clauses and the latter being all other clauses. The *rules* should not be confused with the inference rules. Generally the *rules* are applied to the *facts* in order to produce new *facts*. Initially all hyper extension inference possibilities with premises from *facts* and *rules* are exhausted, and the results end up in *conclusions* a temporary storage. Then a loop begins. The *conclusions* are saved in *newFacts*. All inference possibilities that use at least one of the recently derived result in *newFacts* as a premise are exhausted, with the new results being stored in *conclusions*. At the end of the loop iteration the saved inference results in *newFacts* are appended to the *facts*. The loop restarts, unless a contradiction has been found, or there were no new inference results in the last iteration, indicating that a fixed-point has been reached.

7.3.1 Proof Procedure for Hyper Tableaux

The original hyper tableaux calculus uses a literal tree, and the hyper extension rule derives only literals. For the original implementation in KRHyper this means that that the semi-naive evaluation could be applied to hyper tableaux in a straightforward manner, with *facts* storing the literals and *rules* being the fixed set of clauses. The only major difference is the treatment of splitting: When selecting a non-Horn clause for hyper extension, the resulting positive literals are disjunct, so only one of them can be added to the *conclusions*, while the others must be stored separately. When a branch is closed or exhausted, the prover backtracks to the most recent split and selects one of the other stored disjunct literals instead. The implementation can thus be seen as having two levels: The upper level is a branching control algorithm that handles splitting and backtracking in the hyper tableau, while the embedded lower level is a semi-naive evaluation loop that computes the linear branch sections between splits.

Both levels also cooperate to ensure fairness by incorporating *iterative deepening*. All inference results have their *weight* compared against a weight limit, and if a result exceeds this limit, then it is discarded and the node of the first such transgression is marked. KRHyper and E-KRHyper offer two ways of measuring the weight. One is to use the *term depth* and one is to use the *term size*.

Definition 7.1 (Term Depth)**.** *The* term depth *is a mapping of terms to the natural numbers. Given a term t, the term depth of t is recursively defined as follows:*

- *If t is a constant or a variable, then the term depth of t is* 1*.*
- *If $t = f(s_1, \ldots, s_n)$ for a function symbol f with arity n and subterms $s_1, \ldots, s_n$, then the term depth of t is* 1 + *the maximum term depth of $s_1, \ldots, s_n$.*

The *literal depth* of a literal is determined by treating its atom as a term (i.e. the predicate symbol as a function symbol) and measuring the term depth. The *clause depth* of a clause is equal to the maximum literal depth of its literals.

The term size on the other hand counts the number of symbol occurrences.

Definition 7.2 (Term Size)**.** *The* term size *is a mapping of terms to the natural numbers. Given a term t, the term size of t is recursively defined as follows:*

- *If t is a constant or a variable, then the term size of t is* 1*.*
- *If $t = f(s_1, \ldots, s_n)$ for a function symbol f with arity n and subterms $s_1, \ldots, s_n$, then the term size of t is* 1 + *the sum of the term sizes of $s_1, \ldots, s_n$.*

The *literal size* of a literal is determined by treating its atom as a term (i.e. the predicate symbol as a function symbol) and measuring the term size. The *clause size* of a clause is equal to the maximum literal size of its literals.[7] Generally the size is the preferred weight measure, rather than the depth. The size corresponds more closely to the number of rewritable positions in a term and is therefore a more reliable indicator of the amount of subsequent inferences a result can trigger, an important estimate in equational reasoning. However, when dealing with non-equational problems there is no clear advantage to either method.

Using the weight measure to discard results ensures that the semi-naive evaluation loop terminates when combined with a hyper tableau regularity check, because that way no result can be added more than once, nor can the results grow indefinitely. In other words, the lower level will always compute a finite linear branch segment, and thus it is guaranteed that the upper level takes over eventually. Naturally, discarding results affects the completeness, so if no contradiction is found, branches may have to be recomputed with an increased weight limit, hence the iterative deepening aspect. The details will be explained below.

The algorithm in KRHyper forms the basis for the strategy in E-KRHyper, and to ensure backwards compatibility E-KRHyper can fall back to this original strategy. Thus it remains an integral part of the current prover and is worth a detailed description. Algorithm 7.2 depicts the lower level of the original KRHyper strategy in pseudo-code. The similarities to the basic semi-naive evaluation should be obvious, and the only significant difference is an initial distinction between the root node and split nodes further down in the hyper tableau. In the former case the *facts* and *rules* are initialized by the input problem and all inference possibilities between these sets and within the *weightLimit* are exhausted, while for a split node the *facts* and *rules* are already non-empty, and only the split literal labeling the node is used for the initial inferencing. After that the KRHyper loop closely follows the semi-naive evaluation loop. A difference is that non-unit results are stored separately in *disjunctions*. These do not participate in the reasoning right away, they will rather be used for splitting in the upper level algorithm, as shown in Algorithm 7.3.

Given a *node*, first an attempt is made to extend the hyper tableau at this *node* in a linear manner without splitting, by calling up the lower level semi-naive evaluation. If this terminates, then with one of two possible results, stored in *branchResult*. If the branch has been *closed*, there may still be unprocessed

[7] Intuitively it might make more sense to compute the clause size as the sum of its literal sizes, but in practice this has proven less useful, as it tends to lead to excessive clause size measures.

Algorithm 7.2 The lower level in KRHyper uses semi-naive evaluation to compute the linear branch segments between splits.

```
function EVALUATE_BRANCH_AT_NODE(node)
    if node = rootNode then
        facts := unit clauses from input reasoning problem;
        rules := input reasoning problem \ facts;
        conclusions := all inferences with rules and facts;
    else
        conclusions := {split literal labeling node}
    end if
    while (conclusions ≠ {}) ∧ (contradiction not found) do
        newFacts := unit clauses from conclusions;
        disjunctions := disjunctions ∪ (disjunctions from conclusions);
        conclusions := all inferences with newFacts,
                       using other premises from rules ∪ facts;
        conclusions := conclusions \ results above weightLimit;
        facts := facts ∪ newFacts
    end while
    if contradiction found then
        return closed
    else
        return exhausted
    end if
end function
```

branches that resulted from splitting. If this is the case, the first such split node is taken from the *unprocessedSplitStack*, *facts* and *disjunctions* are backtracked to their appropriate state for the split node, and the computation of the new branch begins by a recursive call to the upper level algorithm. If on the other hand no unprocessed split branches remain, then the hyper tableau is closed, and KRHyper finishes the computation with a *closed* result.

If the lower level semi-naive evaluation indicates an *exhausted* branch in *branchResult*, then there are again different cases determining how to proceed. There may still be unused splitting opportunities that have accumulated in *disjunctions*, and in this case the branch is not yet truly exhausted. Instead one of the *disjunctions* is selected and used for splitting: the disjunctive literals are transferred to the *unprocessedSplitStack* where they represent the unprocessed split branches attached to the leaf of the current branch. The first of these is selected and its computation begins. If on the other hand no open disjunctions are left, then the branch may indeed be exhausted. If no inference result was discarded due to exceeding the weight limit, then a true fixed-point has been reached, the branch being exhausted and representing a model. However, if even one result was discarded, then the algorithm may have missed important inferences that could have closed the branch. In that case the weight limit is increased, and the branch is recomputed starting from the node where the first overweight result occurred.

Algorithm 7.3 At the upper level the KRHyper branching control loop handles splitting and backtracking.

```
function EXTEND_TABLEAU_AT_NODE(node)
    branchResult := EVALUATE_BRANCH_AT_NODE(node);
    if branchResult = closed then
        if unprocessedSplitStack ≠ {} then
            nextNode := pop(unprocessedSplitStack);
            facts := backtrack(facts, nextNode);
            disjunctions := backtrack(disjunctions, nextNode);
            EXTEND_TABLEAU_AT_NODE(nextNode)
        else
            return closed
        end if
    else
        if disjunctions ≠ {} then
            disjunction := take a disjunction from disjunctions;
            push(literals of disjunction, unprocessedSplitStack);
            nextNode := pop(unprocessedSplitStack);
            EXTEND_TABLEAU_AT_NODE(nextNode)
        else
            if weightLimit has been exceeded then
                increase(weightLimit);
                nextNode := node of first weightLimit transgression;
                facts := backtrack(clauses, nextNode);
                disjunctions := backtrack(disjunctions, nextNode);
                EXTEND_TABLEAU_AT_NODE(nextNode)
            else
                return exhausted
            end if
        end if
    end if
end function
```

7.3.2 Proof Procedure for E-Hyper Tableaux

As mentioned above, the aforementioned algorithm also exists in E-KRHyper and can be used for non-equational reasoning problems. However, for the full implementation and usage of the equational E-hyper tableaux calculus some changes are necessary. The basic two-level structure remains, though.

The pseudo-code in Algorithm 7.4 summarizes the lower level evaluation. As the equational calculus can derive new non-unit clauses, the clear distinction between the dynamic *facts* and fixed *rules* no longer applies, and instead one dynamic set of *clauses* is used. Naturally, the inference phases here consist of the rules of the E-hyper tableaux calculus, though as per Proposition 6.1 the hyper extension step is also included as a shortcut inference. Another difference is that inference results exceeding the *weightLimit* are not discarded outright, rather they are saved in the *overweight* set. Finally, reduction phases serve to reduce the amount of clauses in the system by exploiting various redundancy

Algorithm 7.4 The lower level in E-KRHyper uses semi-naive evaluation to compute the linear branch segments between splits.

```
function EVALUATE_BRANCH_AT_NODE(node)
    if node = rootNode then
        clauses := input reasoning problem;
        clauses := reduce clauses by clauses;
        conclusions := all inferences with clauses;
        overweight := inference results above weightLimit;
        conclusions := conclusions \ overweight
    else
        conclusions := {split literal labeling node}
    end if
    conclusions := reduce conclusions by clauses;
    while (conclusions ≠ {}) ∧ (contradiction not found) do
        clauses := reduce clauses by conclusions;
        disjunctions := disjunctions ∪ (disjunctions from conclusions);
        newClauses := conclusions \ disjunctions;
        conclusions := all inferences with newClauses,
                       using other premises from clauses;
        overweight := overweight ∪ (inference results above weightLimit);
        conclusions := conclusions \ overweight;
        clauses := clauses ∪ newClauses;
    end while
    if contradiction found then
        return closed
    else
        return exhausted
    end if
end function
```

criteria. A reduction phase performs numerous operations on each clause of the set to be reduced, most of which are covered by the Del and Simp rules, but which also include special cases of the Equality rules.

Let C be a clause to be reduced and let $\mathcal{S}$ be a set of clauses that can be used to reduce C, then the following reduction operations are attempted exhaustively:

- elimination of every positive literal $A \in C$ for which there is a negative unit $\leftarrow D \in \mathcal{S}$ with $A \gtrsim D$,
- elimination of every negative literal $\neg B \in C$ for which there is a unit $D \leftarrow \in \mathcal{S}$ with $B \gtrsim D$,
- rewriting every subterm $s \in C$ with $r\sigma$ if there is a positive unit equation $l \simeq r \leftarrow \in \mathcal{S}$ with $l \succ r$ and $s \gtrsim l$ (*demodulation)*,
- elimination of negative literals with trivial equational atoms of the form $t \simeq t$,
- elimination of negative literals with equational atoms that have one variable side: if $C = \mathcal{A} \leftarrow \mathcal{B}, x \simeq t$, then replace C with $C' = \mathcal{A}\sigma \leftarrow \mathcal{B}\sigma$ where $\sigma = \{x \leftarrow t\}$,
- limiting multiple occurrences of the same literal to one,
- discarding C if it is non-properly subsumed by some clause $D \in \mathcal{S}$,
- discarding C if it contains a positive literal with a trivial equational atom of the form $t \simeq t$,
- discarding C if it contains the same atom both in a positive and in a negative literal.

The reduction of a clause C thus either results in its minimal canonical form C^{min}, or in the abandonment of C. As C^{min} may turn out to be the empty clause, exception handlers monitor the reduction phases and close the current branch immediately if a reduction to a contradiction is possible.

The E-hyper tableaux calculus only specifies non-proper subsumption, as proper subsumption could negatively affect the model enumeration capabilities. For example, while the clause $C = c(x) \leftarrow$ properly subsumes $D = c(a), d(b) \leftarrow$, a tableau for C alone would only lead to the model $\{c(x)\}$, whereas for C and D together we obtain two models, $\{c(x)\}$ and $\{c(x), d(b)\}$. As it is often more important to solve a reasoning problem (i.e. find one model or one refutation) rather than to enumerate its models, E-KRHyper can optionally also use proper subsumption. Formally this is easily justified by non-proper subsumption and the law of absorption: Recall that a clause C properly subsumes a clause D if $C\sigma \subset D$ for some substitution σ (see Section 2.1.5). Let us decompose D into $D = C\sigma \cup R$, such that R is the set of the other literals in D that are not necessarily instances of literals in C. By the law of absorption lifted to FOL, C is equivalent to the formula $C \wedge (C \vee R')$, where R' is a universally quantified generalization of R that shares no variables with C. $C \wedge (C \vee R')$ can be divided into the two clauses C and $C \cup R'$, and the latter non-properly subsumes D. Thus if C properly subsumes D, then C is equivalent to a clause set that non-properly subsumes D. In practice proper subsumption is difficult to compute, and E-KRHyper usually only tests for proper subsumption when C is a unit, staying with non-proper subsumption for all other cases. See Chapter 10 for more details on the difficulties of clause indexing and subsumption.

The applications of Del and Simp are limited to certain segments of the current branch, specifically to the nodes below the most recent split. E-KRHyper may observe this limitation, but this is optional. The calculus specification considers tableaux as relatively static objects where clauses in a split branch should not make changes higher up that could affect other branches. The backtracking operations in the implementation are capable of undoing such changes, though, so E-KRHyper can eliminate redundant clauses all over the branch.

Algorithm 7.5 At the upper level the E-KRHyper branching control loop handles splitting and backtracking.

```
function EXTEND_TABLEAU_AT_NODE(node)
    branchResult := EVALUATE_BRANCH_AT_NODE(node);
    if branchResult = closed then
        if unprocessedSplitStack ≠ {} then
            nextNode := pop(unprocessedSplitStack);
            clauses := backtrack(clauses, nextNode);
            disjunctions := backtrack(disjunctions, nextNode);
            overweight := backtrack(overweight, nextNode);
            EXTEND_TABLEAU_AT_NODE(nextNode)
        else
            return closed
        end if
    else
        if disjunctions ≠ {} then
            disjunction := take a disjunction from disjunctions;
            push(literals of disjunction, unprocessedSplitStack);
            nextNode := pop(unprocessedSplitStack);
            EXTEND_TABLEAU_AT_NODE(nextNode)
        else
            if weightLimit has been exceeded then
                overweight := reduce overweight by clauses;
                if overweight = {} then
                    return exhausted
                else
                    increase(weightLimit);
                    nextNode := node of first weightLimit transgression;
                    clauses := backtrack(clauses, nextNode);
                    disjunctions := backtrack(disjunctions, nextNode);
                    overweight := {};
                    EXTEND_TABLEAU_AT_NODE(nextNode)
                end if
            else
                return exhausted
            end if
        end if
    end if
end function
```

The changes required by equational reasoning carry over to the upper level algorithm as well, shown in Algorithm 7.5. This remains very similar to its non-equational counterpart, but again it uses a dynamic set of *clauses* instead of the *facts* and a fixed set of *rules*. Another difference lies in the way the *overweight* set affects the computation. When a branch has been exhausted and the weight limit has been exceeded, the algorithm does not immediately backtrack to the node where the first weight limit transgression occurred. Instead the *overweight* clauses undergo a reduction phase. If all of them can be simplified to a point where they are redundant or (non-properly) subsumed by the other *clauses* in the branch, then omitting them from the tableau has not affected completeness. In that case the branch is truly exhausted, and a model can be returned. If on the other hand not all *overweight* results can be rendered redundant, then the weight limit is increased and the tableau is recomputed as usual.

Technically the original KRHyper could have incorporated this treatment of overweight results as well, instead of always simply discarding them and then recomputing the tableau if necessary. One reason against this is that the non-equational hyper tableaux calculus offers no means of rewriting, thus limiting the amount of simplification operations that could reduce the *overweight* set to redundancy. Another one is that both the original calculus and its implementation consider the notion of redundancy only in a rudimentary fashion, as it is not quite such a pressing issue within non-equational FOL. On the other hand the equational reasoning with E-hyper tableaux requires a much more thorough treatment of redundancy, leading to ubiquitous detection and elimination methods throughout E-KRHyper. Finally, one needs to weigh the advantages and disadvantages. Saving and testing overweight results instead of discarding them allows finding more models and thus proving more satisfiable reasoning problems, but at the cost of a higher usage of memory. This trade-off could be called into question depending on the application of the ATP. For example, among the 15,550 FOL problems of the TPTP v5.3.0 there are 12,570 problems known to be unsatisfiable, while only 1,970 are known to be satisfiable. When using a prover in such an environment, the ability to find more models is of dubious value if it is detrimental to the ability to find refutations. In our experiments with the TPTP E-KRHyper solves 4% more problems when not retaining overweight results. Especially in a competitive situation like the CASC with a strong emphasis on unsatisfiable problems it thus makes sense not to burden an ATP with enhancements for model generation. The algorithms shown here therefore represent the default behaviours, and optionally E-KRHyper can be set to discard overweight results like KRHyper. Also, in a similar manner E-KRHyper will try to exploit redundancy even when working with non-equational problems, but it can be configured to behave like KRHyper if this is desired.

7.4 Implementation Details

This section will describe some aspects of the implementation in more detail, either because they may be of general interest for the understanding of E-KRHyper, or because they are relevant for the adaptations to LogAnswer that will be presented in later chapters.

7.4.1 Indexing

A theorem prover has to maintain thousands or possibly millions of clauses during its work on a reasoning problem. The naive solution of simply storing them in a list quickly becomes inefficient when inferences need to search for clauses or terms matching specific criteria. This is a frequent occurrence in the processing of a problem. Given a query Q which may be a term, an atom, a literal or a clause, the different searches can be categorized into four groups:

Generalization searching: search for those A that are generalizations of Q ($Q \gtrsim A$) - used during:

- forward subsumption: is Q (non-properly) subsumed by some existing A?
- forward demodulation: is there an oriented equation $A \simeq r$ with some term $r \prec A$ that can demodulate Q?

Instance searching: search for those A that are instances of Q ($A \gtrsim Q$) - used during:

- back subsumption: does Q (non-properly) subsume existing A?
- back demodulation: if Q is the left side of an oriented equation $A \simeq r$ with some term $r \prec Q$, can this equation be used to demodulate existing A?

Unification searching: search for those A that unify with Q (i.e. there is a σ with $A\sigma = Q\sigma$) - used for:

- hyper extension: are there branch literals A that unify with the selected clause atom Q?
- superposition (unit-sup-right and sup-left): if Q is one side of an equation, can this equation Q be used for superposition on existing A?

Variant searching: search for those A that are variants of Q ($Q \sim A$) - this is rarely necessary, but it can be useful to find out whether some Q has already been stored.

Given the frequency of situations where searching is necessary, and given the possibly large size of the clause sets to search, any list based algorithm like sequential or binary search would be far too inefficient. There are several known methods to solve (or at least alleviate) this problem, and the common solution in ATP systems is to use some form of indexing that allows a significantly faster retrieval of stored data. E-KRHyper is no exception, and it stores the clauses in *discrimination tree* indexes [McC92].

A discrimination tree is a tree whose root node bears the empty label, whose non-root nodes are labeled with signature symbols and a special symbol $\mathbf{X}$, and whose leaves are used to store the indexed data. Let $\mathcal{T}$ be a set of terms to be indexed in a discrimination tree $\mathbf{D}$. Let t be a term, then $f^{\mathbf{X}}$ is defined as a mapping such that $f^{\mathbf{X}}(t)$ is a copy of t where every occurrence of a variable symbol is replaced by $\mathbf{X}$, and the *flattened term flat*(t) is defined as the sequence of symbols derived by a preorder traversal of t. Then each term $t \in \mathcal{T}$ is stored in $\mathbf{D}$ at the leaf node of the branch whose sequence of node labels from root to

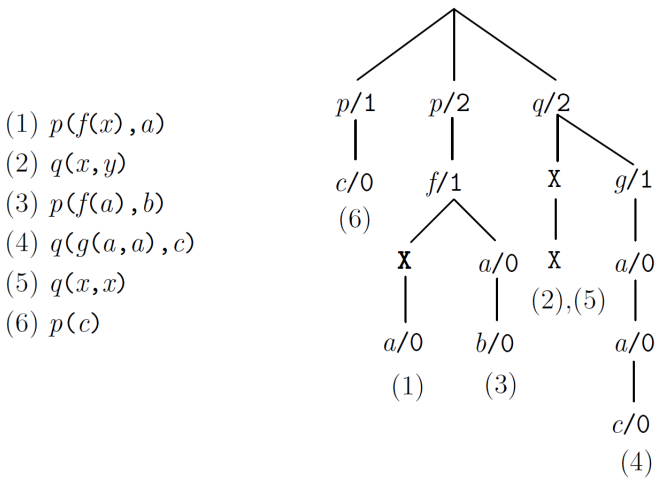


Figure 7.2: Discrimination tree indexing the terms (1) - (6) (the labels carry symbol arity information for clarity)

leaf corresponds to $f^{\mathbf{X}}(\mathit{flat}(t))$. Note that this may entail storing multiple terms at one leaf due to them only differing in their variables. Figure 7.2 illustrates the concept with an example.

When searching for terms in the discrimination tree index, the query term is flattened and the resulting symbol sequence guides the search from the root to a number of leaves, always selecting the node whose label matches the next symbol in the sequence. Whenever the search reaches a leaf, all the terms stored there are returned as results. The search then continues to seek for the next leaf until all matching possibilities have been exhausted. As different variables are not distinguished in the tree labeling, they require special treatment depending on the specific search category.

Let q be a query term and let s be the current symbol in q that has to be matched in a discrimination tree $\mathbf{D}$. Let q^s be the subterm in q headed by s. Let q^t be the subterm following q^s in q with t being the symbol heading q^t.

Generalization searching:

- If s is a variable it matches the node label $\mathbf{X}$, and
- otherwise s matches the node labels
  - s, and
  - $\mathbf{X}$, and in this case the remaining symbols of q^s also match, so the symbol arities are used to skip ahead in q to t.

Instance searching:

- If s is a variable it matches any node label l, and if the arity of l is larger than zero, then the search skips ahead in the tree to those nodes that are the appropriate arity-determined skipping distance away, and
- otherwise s matches the node label s.

Unification searching:

- If s is a variable it matches any node label l, and if the arity of l is larger than zero, then the search skips ahead in the tree to those nodes that are the appropriate arity-determined skipping distance away, and
- otherwise s matches the node labels
  - s, and
  - $\mathbf{X}$, and in this case the remaining symbols of q^s also match, so the symbol arities are used to skip ahead in q to t.

Variant searching:

- If s is a variable it matches the node label $\mathbf{X}$, and
- otherwise s matches the node label s.

These search algorithms do not actually test whether the variable bindings are free of clashes, so the retrieved terms are not certain to meet all the desired criteria with respect to the query term. They can merely be viewed as candidates, and an additional test (unification, variant, or subsumption in the respective direction) is required to filter out the final results.

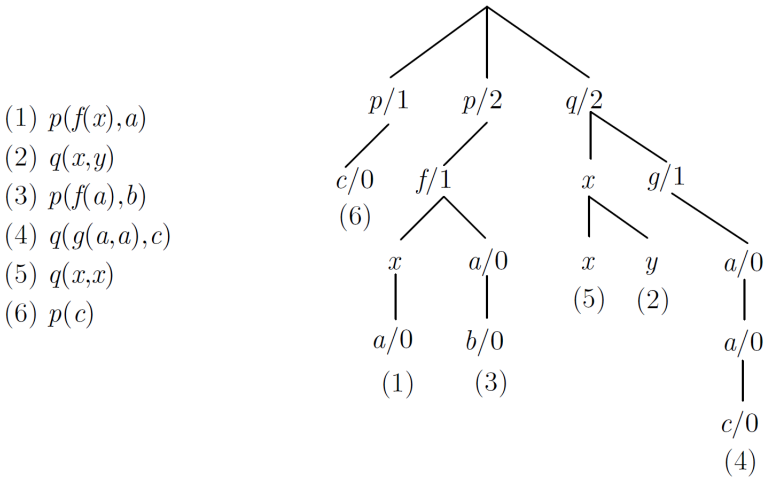


Figure 7.3: Perfect discrimination tree indexing the terms (1) - (6)

This does not have to be the case. A variant of this indexing is known as *perfect discrimination trees* (and in contrast the ordinary discrimination trees are sometimes referred to as *imperfect* discrimination trees). Perfect discrimination trees differ from the imperfect ones in that they do not use the special label $\mathbf{X}$ to represent all variables. They forgo using the $f^{\mathbf{X}}$ function and instead form the flattened term $\mathit{flat}(t)$ directly based on t. The result is that distinct variables are represented by distinct labels, as shown in the example in Figure 7.3.

While this has a disadvantage in that the index may grow considerably larger, its advantage lies in that the search algorithms can attempt to compute the

appropriate substitutions on the fly and detect a clash before a leaf is reached. Once a leaf is reached, the retrieved results are guaranteed to match the search criterion, so unlike the imperfect candidates they do not require further testing. While the original KRHyper only needed imperfect discrimination trees to store the branch literals, the use of indexing has greatly expanded in E-KRHyper, and both imperfect and perfect discrimination trees are employed depending on their specific usage.

Several aspects of the indexing in E-KRHyper will now be described in more detail.

Subterm Indexing

The indexing in the original KRHyper is only used to store atoms and literals, and the flattened terms, which determine the positions within the index trees and the search paths, are always based on atoms. The first symbol in the symbol sequences is therefore always a predicate symbol, and the leaves always store atoms or literals. In E-KRHyper this is not necessarily the case. The superposition-based inferences need to access the subterms of clauses, and given an equation $l \simeq r$ E-KRHyper must then find clauses containing a subterm s that unifies with l. This means that E-KRHyper needs to index clauses by their subterms. Every non-variable subterm in a positive unit clause and every non-variable subterm in a body literal is open for rewriting, so the respective clauses must be accessible in the index by these subterms. If the optional demodulation is in use, the subterms of head literals in non-unit clauses are also made accessible in a separate index. Due to the large overall number of subterm positions, the subterm indexes are implemented as imperfect discrimination trees in order to avoid excessive branching.

Let A be the atom of an indexable literal in a clause C. Then $A = p(t_1, \ldots, t_n)$ for some predicate symbol p with arity n and subterms $t_1, \ldots, t_n$, with $n \geq 0$. Then the subterm index discrimination tree contains an entry for each non-variable $t \in \{t_1, \ldots, t_n\}$, with the indexing branch determined by $\mathit{flat}(f^{\mathbf{X}}(t))$ and C stored at its leaf. The indexing is recursive to cover all non-variable subterms, so if $t = f(t'_1, \ldots, t'_m)$ for some function symbol f with arity $m > 0$, then the discrimination tree also contains entries storing C at the leaves of the pathes determined by $flat(f^{\mathbf{X}}(t'_i))$ for all $1 \leq i \leq m$, and so on.

Figure 7.4 demonstrates the principle with an example. The unit clause $(1) = p(f(a, x)) \leftarrow$ is indexed by the subterms $f(a, x)$ and a, while clause $(2) = q(b, c) \leftarrow q(a, g(x)), h(y) \simeq a$ is indexed by a, $g(x)$, $h(y)$ and once more a. With a occurring twice in clause (2) there are also two index entries for that clause at the end of the path determined by a.

As the example shows, a clause may be stored multiple times at the same leaf. This is not redundant, as the subterm index does not actually store just the clauses at the leaves. Rather, each leaf entry is a data tuple containing the indexed clause and additional information, including a pointer to the exact position of the indexing subterm in the clause. When a superposition inference has retrieved a clause for rewriting, it can use the pointer to directly access the potentially rewritable subterm. It does not have to search the clause itself again to find the indexed position. The double entries in the example would thus differ in their subterm pointers. Of course all index entries of a clause also share the clause itself, so while the indexing can result in a large number of

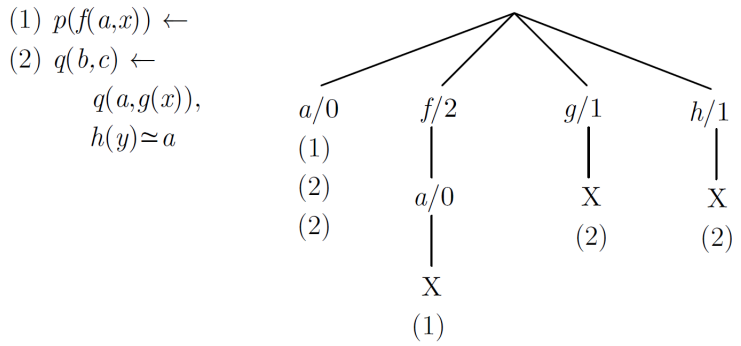


Figure 7.4: Discrimination tree indexing by subterms for clauses (1) and (2)

entries for a single clause, these mostly consist of references to the clause, which itself is stored only once in the memory.

Equation Indexing

The discrimination trees as described always index by a uniquely determined sequential flattened term, regardless of whether it is based on an atom or a term or subterm. This poses a problem when equational literals are to be indexed by their atoms, as the equality symbol is commutative and it is not clear whether an equation $l \simeq r$ should be indexed by $\mathit{flat}(l \simeq r)$ or by $\mathit{flat}(r \simeq l)$. The term ordering could be used to prefer a canonical form in some cases, but l and r may be incomparable and the equation thus not orientable. The solution in E-KRHyper is therefore to index equations twice, covering both orientations. While this indexing method obviously results in double entries and two index insertion operations per equation, the alternative would have been to index only by one orientation and then to do all searches in double, for both orientation possibilities. As searches are more frequent than insertions, the current solution seems preferable.

Indexing Clauses with Multiple Literals

The indexing cases so far index by a flattened term based on a single atom or term, which is sufficient for most of the inference rules of the hyper tableaux calculi: The hyper extension needs to search for literals, and the superposition-based rules search for subterms. However, to determine non-proper subsumption for more than just unit clauses it is necessary to search for clause pairs that match in multiple literals simultaneously. This is a hard problem, because the associativity and commutativity of the disjunction $\vee$ and the commutativity of equations combine to allow up to $n! \cdot 2^n$ permutations of every clause with n literals, and it is not feasible to index all of these. The method we devised for E-KRHyper is intended to handle both forward and backward non-proper subsumption on large clause sets as encountered in the context of QA. To the best of our knowledge our solution is unique, and as such it warrants the detailed description that forms Chapter 10.

Layered Indexing

When a prover backtracks to an earlier choice point within a derivation, any terms, literals and clauses indexed during the computation of the backtracked part of the derivation become invalid, and they must be removed from the index. The E-hyper tableaux calculus and E-KRHyper are no exception to this, and whenever a branch is closed and E-KRHyper moves on to a different split branch, the clauses of the closed branch up to the parent node of the new branch must be discarded. A common solution would be to backtrack the index as well, by seeking out the respective index entries and deleting them from the leaves, and ideally also deleting subtrees if they no longer contain any entries.

E-KRHyper on the other hand uses a different method that was already employed by the original KRHyper: Each index is arranged as a *layered index*. This means that it does not use a single discrimination tree to store all entries for a branch. Instead, whenever a tableau branch is split, another discrimination tree is added, forming a new *layer*. A branch with n decision nodes is therefore represented by the $n+1$ discrimination trees $T_1, \ldots, T_{n+1}$. New inference results are stored in the most recent layer T_{n+1}. Index searches on the other hand are performed sequentially on all layers. This is implemented in such a way that the layering aspect is hidden from the inference level: A search only has to be started once, and it will automatically visit all layers and gather the results, returning them in a single list.

When backtracking to the i-th choice point in the branch, the references to the layers $T_{i+1}, \ldots, T_{n+1}$ are simply dropped, thereby making them inaccessible to insertion and retrieval operations and leaving their memory to the OCaml garbage collector.

Whether this behaviour is generally preferable is not clear. While backtracking is greatly simplified, the complexity of search operations is multiplied by n. Hence layered indexing is likely to be more efficient on shallow tableaux with a high branching factor, thus keeping the multiplier n low while making ample use of the cheap backtracking. On deep tableaux though the reduced search speed may offset any gains from the enhanced backtracking. However, layered indexing will become highly advantageous in an adaptation to QA that will be described in Section 11.2.

7.4.2 Disjunction Handling

The algorithms in Section 7.3.2 show how the proof procedure of E-KRHyper attempts to postpone splitting by collecting splitting opportunities in a set of *disjunctions*, and only applies a disjunction in a Split inference when no linear branch extension is possible.

There is an exception to this. If a branch contains negative unit clauses that contradict all literals of a purified disjunction D, then splitting with D would only result in split branches that could be closed immediately. Postponing such a split is therefore not optimal. In the original KRHyper the *disjunctions* remain largely removed from inferencing until splitting becomes unavoidable, which means that considerable time can pass even though splitting and closing would be possible. In E-KRHyper however the *disjunctions* use a special index, and whenever a negative unit clause N is derived, the index tests whether it contradicts any hitherto uncontradicted literals of any *disjunctions*. If this is the

case, then N is stored with those literals, and the algorithm examines whether any affected disjunction D now has contradicting units stored for all its literals. If so, then D is immediately selected for splitting and closing the branch.

Even if there are not sufficient negative units to completely negate a saved disjunction, the contradiction index is still useful when picking a disjunction for splitting, as it provides a quick way to determine which disjunctions will result in the least number of new branches.

When disjunctions need to be purified, the purifying terms are obtained from the Herbrand universe that is enumerated using *domain clauses* added to the reasoning problem if necessary:

- $dom(a) \leftarrow$
  for every constant a in the input (add a if there is none),
- $dom(f(x_1, \ldots, x_n)) \leftarrow dom(x_1), \ldots dom(x_n)$
  for every n-ary function symbol f.

The enumeration of the Herbrand universe is postponed as long as possible. It is only started when E-KRHyper has no other choice than to split on an impure disjunction. This is an advantage over KRHyper which requires all non-Horn input to be range-restricted [MB88] and which always begins enumerating the Herbrand universe immediately for such reasoning problems, just in case this is needed later on. This can lead to en explosive generation of domain units, which is a wasted effort if the reasoning problem can be solved without actually splitting on any of its impure disjunctions.

7.4.3 Redundancy Handling

The Del and Simp rules allow destructive tableau modifications by replacing or deleting clauses that meet some redundancy[8] criterion. For practical purposes even replacing involves deletion: While the Simp rule is depicted to replace a node label, the structures representing clauses and nodes are so complex that it is inefficient to relabel a node, and it is better to add the replacing clause as a new leaf node and to delete the old clause.

Technically E-KRHyper could remove redundant clauses from its indexing structure, but we found this to be impractical. Rather, such clauses will be marked as redundant, which prevents them from participating in most inferences. This is preferable for three reasons.

Firstly, given the subterm indexing a clause may be found at a large number of leaves in the discrimination trees, and the time spent retracting all those entries when actually deleting a redundant clause can offset the gains from no longer having to consider the clause during reasoning. In this regard a deleting solution also appeared to go against the spirit of the layered indexing, which specifically serves to avoid retracting separate discrimination tree entries. However, this particular concern may be more about a break in style than about real efficiency.

Secondly, the introduction of a marking mechanism makes it easy to reintroduce formerly redundant clauses by unmarking them. While this is not necessary

[8]Keep in mind that we use the notion of redundancy in a general sense here, which includes both the explicitly defined redundancy of the E-hyper tableaux calculus (see Definition 6.3) and non-proper subsumption (see Definition 6.4).

in an exact implementation of the E-hyper tableaux calculus which restricts destructive tableau modifications to the branch segment below the lowest split, it gives E-KRHyper the optional ability to exploit redundancy across the entire branch, as any modifications above the lowest split can simply be undone during backtracking by unmarking the respective clauses again.

Finally, as redundancy is often difficult to detect, keeping a redundant clause in the system prevents it from reappearing in a seemingly non-redundant form by other means. For example, consider these clauses:

C: $p(f(a)) \leftarrow$

D: $f(x) \simeq x \leftarrow$

E: $p(f(x)) \leftarrow p(x)$

Here D can derive $p(a) \leftarrow$ from C using unit-sup-right, making C redundant in the process. Now with $p(a) \leftarrow$ and E it is again possible to derive $C' = p(f(a)) \leftarrow$, the same clause as C. Had C been deleted due to redundancy, then $C' = C$ would be introduced again as non-redundant, and in this particular example this would even lead to a cycle. If on the other hand C remains in the system and is simply marked as redundant, then its presence blocks C' during a check for regularity or non-proper subsumption. This assumes of course that the implementation allows marked clauses to subsume others, even though they are otherwise forbidden to act as inference premises.

7.4.4 Clausification

The clausifier which transforms input formulas into clause normal form uses the following basic steps in the order established by the Otter ATP system:

1. eliminate the shorthand notations for implications ($\rightarrow, \leftarrow$) and equivalences ($\leftrightarrow$),
2. move negations in so that the symbol $\neg$ only occurs above atoms (*negation normal form* (NNF)),
3. universally quantify free variables,
4. rename variables so that no variable name is quantified more than once,
5. eliminate existential quantifiers via Skolemization,
6. distribute conjunctions and disjunctions so that each formula is a conjunction of disjunctions (*conjunctive normal form*),
7. turn every conjunct into a clause.

A straightforward implementation of step 6 can result in an exponential increase in the number of generated clauses that renders the clausification of deeply nested formulas unfeasible. E-KRHyper prevents this by using *formula renaming* [NW01], a satisfiability preserving technique that replaces a subformula by a literal "pointing" to the replaced subformula in a new formula.

Generally given a Skolemized NNF formula F of the form $G \vee H$, the subformula H can be renamed by replacing F with the formulas

F_1**:** $G \vee P(x_1, \ldots, x_n)$

F_2**:** $\neg P(x_1, \ldots, x_n) \vee H$

where P is a new predicate symbol and $x_1, \ldots, x_n$ are the variables of H. If G and H are conjunctions with m and r literal conjuncts respectively, then fully distributing F will result in $m \cdot r$ clauses, while distributing the replacement formulas F_1 and F_2 merely generates $m + r$ clauses. The decision when to use formula renaming can have a significant effect on the number of generated clauses and the manageability of the resulting clause set by a given calculus. Excessive renaming can be detrimental. The original definition [NW01] proposes a scheme based on formula polarity and a coefficient based estimate of the number of generated clauses. This is implemented in the prover E, which can be used as a stand-alone clausifier. We experimented with this and found that a simpler scheme gave slightly better results in E-KRHyper. For this we again regard Skolemized NNF formulas, which we keep fully flattened in that there are no conjunctions (disjunctions) immediately below conjunctions (disjunctions); for example, instead of $A \wedge (B \wedge C)$ we write $A \wedge B \wedge C$. Also, if a formula is a conjunction at its top level, then we split it into its conjuncts, regarding each as a separate formula. Thus every formula is either a literal or it is a disjunction whose disjuncts are literals or conjunctions of subformulas, and it can be written as $L_1 \vee \cdots \vee L_m \vee C_1 \vee \cdots \vee C_n$, where $L_1, \ldots, L_m$ are literals and where $C_1, \ldots, C_n$ are conjunctions with at least two conjuncts each. Given such a formula we distribute C_1 and apply formula renaming to $C_2, \ldots, C_n$. The resulting formulas are again flattened, split by their conjuncts and then distributed and renamed in the same manner, until all are in CNF.

Overall this simpler uniform method results in a comparable number of clauses while being faster to compute. For example, E-KRHyper can clausify the TPTP version of Cyc in 97.6 seconds, resulting in 3,341,985 clauses, whereas E needs 233.2 seconds to generate 3,341,962 clauses. Over the entire TPTP E-KRHyper solves about 5% more problems when using its own clausifier compared to employing E. However, note that such differences are also dependent on the calculus, since the slightly differing clause sets may be more or less ideal for a given calculus - the prover E-Darwin (see Section 9.1) solves more problems with E than with its own clausifier, which is identical to the one in E-KRHyper.

7.5 Evaluation

For a general evaluation we tested the current E-KRHyper 1.3 on the 15,550 FOL problems of the TPTP v5.3.0. The test system featured an Intel Q9950 CPU with four cores running at 2.83 GHz. E-KRHyper always uses a single CPU core to process one problem, thus allowing four instances of E-KRHyper to work on four problems simultaneously. Every such instance was limited to 1 GB of RAM, and the time per problem was limited to 300 seconds. This setup is comparable to the official TPTP tests. E-KRHyper solved 5,986 problems, corresponding to 38.5% of the test set, or 30.8% of the total TPTP with 19,446 problems. The hardest problem solved was *SET990+1* with a rating of 0.96.

In the official TPTP test result listings[9] E-KRHyper is listed in its 1.2 release from 2011. This version solved 33% of the tested 15,550 FOL problems or 27% of the total TPTP, which means it was the 12th best system out of 62, or the 10th best when accounting for systems sharing positions due to having the same performance. All else remaining equal the current E-KRHyper 1.3 could expect to reach the 9th position, or the 8th when accounting for shared rankings. Note though that the majority of the systems is specialized, and while all of them have their total solving rate with respect to the TPTP listed, they are usually only tested on subsets smaller than the FOL problems. Only the top 20 systems have been tested on more than 15,000 problems and can thus be seen as generalists, so E-KRHyper occupies a middle position among these.

As E-KRHyper is intended to supplant the original KRHyper, we also tested the final KRHyper 5.4.6 under the same conditions as above. KRHyper has no native equality handling, so when necessary we added the required equality axioms to problems with equality. The prover solved 3,756 problems, which is 24.2% of the FOL test set or 19.3% of the total TPTP. The hardest problem solved was *PUZ050-1* with a rating of 0.94.

Comparing the systems in more detail, E-KRHyper solved 3,490 of the 11,626 problems with equality, corresponding to 30%, while KRHyper only solved 1,548 of these problems, or 13.3%. This difference is easily explained by the lack of native equality handling on the calculus level in KRHyper. More interesting is the difference regarding the 3,924 problems without equality. Here E-KRHyper solved 2,496 problems, or 63.6%, while KRHyper solved 2,208 problems, or 56.3%. Since E-KRHyper essentially falls back to the basic hyper tableaux calculus when dealing with problems without equations, this performance gap must be attributed to general implementation improvements in E-KRHyper and the adaptation of redundancy detection routines to the processing without equality.

E-KRHyper has participated in CASC since 2007, the first in the KRHyper family of provers to do so. The competition performance over the years reflects the general improvements to E-KRHyper. In 2007 E-KRHyper had not yet been tested extensively. The system could only process input in the PROTEIN syntax and lacked a clausifier. Therefore it only took part in the CNF-based categories, with the problems having been converted into PROTEIN. We do not wish to repeat the extensive result tables found at the website[10] for the CASC of 2007 [Sut08], so suffice it to say that the early E-KRHyper performed quite poorly. The Otter system always participates in CASC as a benchmark due to its long history, stability and decent performance. E-KRHyper mostly remained below Otter in 2007. Its best performance was in the HNE category (Horn problems without equality) where it reached the sixth position out of ten.

In 2008[11] [Sut09] E-KRHyper participated in more categories, because now it had gained support for TPTP syntax and a clausifier. The prover still remained below Otter in the generalist CNF and FOF divisions. Its best positions were fifth out of eight in EPS (effectively propositional satisfiable problems) and sixth out of 11 in NNE (non-Horn without equality).

[9] Most recent result listing: `http://www.cs.miami.edu/~tptp/TPTP/Results.html`
Our data is based on the results retrieved on 1 May 2012.

[10] `http://www.cs.miami.edu/~tptp/CASC/21/`

[11] `http://www.cs.miami.edu/~tptp/CASC/J4/`

In 2009[12] [Sut10] the teething troubles of integrating E-KRHyper into LogAnswer had been solved, so now more effort had been put into a general performance improvement. Thus for the first time E-KRHyper exceeded the Otter benchmark in the categories CNF and FOF. The prover reached the fifth position in several divisions and categories, including EPS and SAT (satisfiable clause sets), as well as the FNT division (first-order form non-propositional non-theorems) The participation in the LTB division was effectively futile, as E-KRHyper did not yet have any axiom selection algorithms suitable for the size of these problems.

In 2010[13] [Sut11] the performance of E-KRHyper remained roughly the same among a larger field of participants. The best position was sixth out of 13 in HNE (Horn without equality). In this category E-KRHyper reached the highest SOTAC score of all participants, meaning it solved more hard problems than the others. E-KRHyper did not participate in LTB due to the experiences of the previous year and the lack of improvements specifically for this division.

In 2011[14] [Sut12] E-KRHyper featured additional performance improvements. In the FOF division it outclassed the LEO-II and Metis systems which had been superior to E-KRHyper in the previous year. It is also noteworthy that E-KRHyper performed better than our own E-Darwin in several categories, despite E-Darwin normally solving more problems, although this is likely in part to blame on a troublesome experimental change to E-Darwin at the time. The best performance of E-KRHyper was in the FNQ category (first-order form non-propositional non-theorems with equality), where it reached the fourth position out of eight and the highest SOTAC score of all participants. This time E-KRHyper participated in the LTB division, as it had been equipped both with appropriate axiom selection algorithms (see Section 12.2.2) and a control script to process series of problems with shared axiom sets. Unfortunately, despite extensive testing before CASC the participation was marred by technical failures, the reason for which could only be identified afterwards: The CASC problems were taken from the TPTP v5.3.0, at the time still unreleased, and some of the axiom sets in this version had been revised to include very large numerical constants. As E-KRHyper features arithmetic evaluation, it tries to represent such constant names not merely as strings, but as floats or integers respectively. However, the new numerical constants were so large that they exceeded the system limits for these numbering formats, causing E-KRHyper to crash. This flaw has since been remedied.

Overall the history of CASC participations shows E-KRHyper moving from a bottom-ranked system to a middle position. There is still room for future improvements. Eventually though some fundamental properties of E-KRHyper form a performance ceiling which will prevent it from becoming a top-ranked prover, see Section 9.2. Changing these properties would go against the original purpose of E-KRHyper, so for reasons of backwards compatibility to existing applications this is unlikely to happen.

This closes the system description of E-KRHyper. The next chapter will give an overview of the QA system LogAnswer, in which I have embedded E-KRHyper. Subsequent chapters are then dedicated to particular modifications and adaptations of E-KRHyper to LogAnswer.

[12] `http://www.cs.miami.edu/~tptp/CASC/22/`
[13] `http://www.cs.miami.edu/~tptp/CASC/J5/`
[14] `http://www.cs.miami.edu/~tptp/CASC/23/`

Chapter 8

The Question Answering System LogAnswer

This chapter provides an overview of the QA system LogAnswer [7]. It will not be as detailed as the previous system description of E-KRHyper, since unlike the prover the specifics of the various other subsystems were neither part of my work, nor are they in the focus of this dissertation. The intention is to achieve a general understanding of the framework in which E-KRHyper is embedded, so that the adaptations of the prover make sense to the reader.

8.1 Background and Development History

The basic LogAnswer system prototype was quickly assembled during the early stages of the project in 2007 [6]. This was only possible because many of its components were already well established, like the earlier mentioned WOCADI parser, the IRSAW information retrieval system and the HaGenLex computational lexicon. LogAnswer was therefore able to participate already in the CLEF QA competition of 2008 [9]. Throughout the course of the project the system has been constantly refined as our understanding of the pertinent issues increased, and over the years we have extended LogAnswer with new capabilities in order to explore the possibilites of logic-based QA.

The main purpose of LogAnswer is to support research in the combination of automated reasoning and question answering. This is reflected in the design, which allows scaling and adapting the system to different use cases. These range from the usage as a search engine replacement, requiring minimal response times, to scenarios with generous time limits which allow deep reasoning, for example in research tools for scientists, or in the competition situation at CLEF, the latter being the primary opportunity for an independent evaluation of LogAnswer.

The system is intended for open-domain question answering. This means it is not specialized in any particular topic, and instead it aims at answering arbitrary questions. Obviously this means that LogAnswer needs access to a vast amount of knowledge, and this is ensured by a knowledge base derived from the German Wikipedia.[1] With gradual improvements in the automatic

[1] `http://de.wikipedia.org`

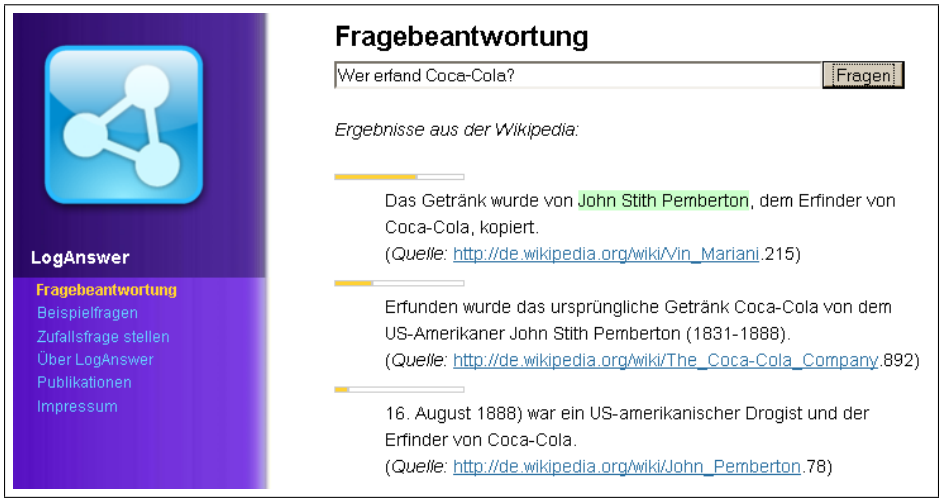


Figure 8.1: Screenshot of the LogAnswer web interface

translation, the knowledge base has almost tripled in size over the years, and it contains data extracted from about 29 million sentences at the time of this writing.

Currently LogAnswer is restricted to German language questions. This is because some of the NLP tools involved in LogAnswer were developed specifically for German. However, in general our approach could be extended to other languages as well. For consistency with the rest of the dissertation examples will thus usually be given in English, or in some cases in the original German together with English translations.

8.2 Usage Information

Like most QA systems LogAnswer requires an extensive hardware infrastructure, and thus the system is not easily portable. It can be accessed by different means, though, of which the browser-based web interface[2] is the most elaborate and mature. Figure 8.1 shows a screenshot of the QA screen as presented in the browser. The user can enter a question or choose an example question. After a few seconds LogAnswer will show up to five answers, sorted by a quality estimate. A bar indicates LogAnswer's confidence in each answer. The exact answers are highlighted in green within their source sentences. In cases of low confidence an answer may only consist of such a sentence. This is the case when no exact answer could be extracted, yet heuristics considered the sentence as a whole to be relevant.

Apart from the website there is also an application for the iPhone[3] and an experimental SMS-based service. Both are essentially user interfaces to the main LogAnswer server, providing an access alternative to the LogAnswer website, and they demonstrate the usage possibilities on small mobile devices. For com-

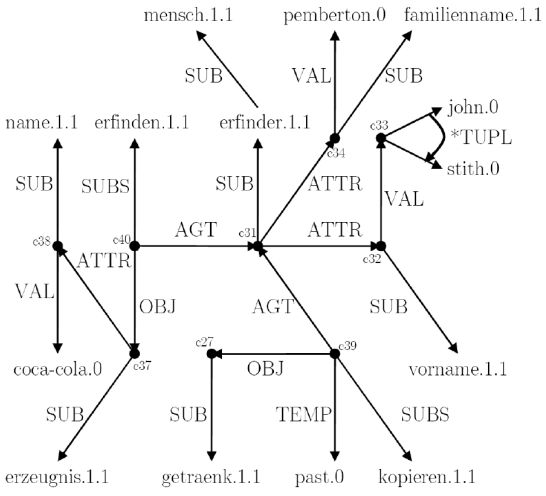
[2] `http://www.loganswer.de`
[3] `http://itunes.apple.com/app/loganswer/id383308349`

petitions LogAnswer does not rely on any user-oriented interface, instead the questions are provided in files directly on the LogAnswer hardware, to interfaces that are for internal use only.

8.3 Knowledge Representation in LogAnswer

The knowledge base of LogAnswer represents knowledge using the MultiNet formalism, an extended form of semantic networks that was designed to capture the meaning of natural language. MultiNet features a stable and relatively compact inventory of relations that label the network edges. Its expressivity goes beyond traditional semantic networks due to the use of *layer attributes* that handle quantification and other aspects of natural language that are difficult to express with relational means. A set of inference rules formalizes the semantics of words and the logical properties of the MultiNet relations; these rules allow reasoning on MultiNet networks.

For the processing with the embedded E-KRHyper the MultiNet knowledge must be further translated into first-order logic, specifically the TPTP format. The MultiNet nodes and attributed arcs can be translated in a fairly straightforward manner into constants and predicates respectively, and the same holds for the logical MultiNet inference rules.



(a) MultiNet

sub(*erfinder.1.1*, *mensch.1.1*)
∧ subs(*c40*, *erfinden.1.1*)
∧ obj(*c40*, *c37*)
∧ agt(*c40*, *c31*)
∧ temp(*c39*, *past.0*)
∧ subs(*c39*, *kopieren.1.1*)
∧ agt(*c39*, *c31*)
∧ obj(*c39*, *c27*)
∧ sub(*c38*, *name.1.1*)
∧ val(*c38*, *coca-cola.0*)
∧ sub(*c37*, *erzeugnis.1.1*)
∧ attr(*c37*, *c38*)
∧ val(*c34*, *pemberton.0*)
∧ sub(*c34*, *familienname.1.1*)
∧ *tupl(*c33*, *john.0*, *stith.0*)
∧ sub(*c32*, *vorname.1.1*)
∧ val(*c32*, *c33*)
∧ sub(*c31*, *erfinder.1.1*)
∧ attr(*c31*, *c34*)
∧ attr(*c31*, *c32*)
∧ sub(*c27*, *getraenk.1.1*)

(b) First-order logic

Figure 8.2: Example for the translation from natural language over MultiNet to first-order logic, for the NL-sentence *"Das Getränk wurde von John Stith Pemberton, dem Erfinder von Coca-Cola, kopiert."* (*"The beverage was copied by John Stith Pemberton, the inventor of Coca-Cola."*)

Figure 8.2 shows an example of the translation of a natural language sentence into MultiNet and then into FOL. The German sentence is *"Das Getränk*

wurde von John Stith Pemberton, dem Erfinder von Coca-Cola, kopiert." (*"The beverage was copied by John Stith Pemberton, the inventor of Coca-Cola."*) The figure is simplified to improve legibility. It contains only the fragment of the representation which models the relational structure of the network. The integer suffixes of the word-based constants are used for disambiguation. They identify the particular word sense represented by the word within the current context, distinguishing for example between nouns and verbs that happen to have the same word form. A subnet of the network may illustrate the MultiNet approach: The node labeled *c31* represents the person John Stith Pemberton. As such it is subordinate to the concept *erfinder* (*inventor*), expressed by a SUB arc to the node *erfinder.1.1*, which in turn is subordinate to *mensch* (*human being*). *c31* also has attributes represented by the nodes *c32* and *c34* (connected by ATTR arcs). The latter is subordinate to *familienname* (*surname*) and has the value *pemberton* - it represents the surname *Pemberton*. *c32* is the analogous for the given names *John* and *Stith*. The special *TUPL arc states that the two given names form the value of node *c32* in that particular order; i.e. it expresses that the person's name is *John Stith Pemberton*, not *Stith John Pemberton*.

The logical background knowledge contains axioms like the following:

$$\forall H \forall P \forall T : ((\text{hsit}(H, P) \wedge \text{temp}(H, T)) \rightarrow \text{temp}(P, T))$$

This example expresses the transitivity of the temporal relation: If the situation H takes place within the temporal frame T ($\text{temp}(H, T)$) and if H is a *hypersituation* with respect to situation P ($\text{hsit}(H, P)$, i.e. P is a part of H), then P also takes place within T ($\text{temp}(P, T)$).

Not all aspects of MultiNet are as easily translated into FOL. Some of the representational means exceed the expressivity of pure first-order logic. As E-KRHyper features logic extensions like arithmetics and list expressions, some of the more difficult MultiNet expressions can be translated into extended FOL representations with equivalent semantics by using the special predicates compatible with the prover. Others, for example generalized quantifiers like *"most"* or *"almost all"* are lost in the translation to logical expressions.

8.4 Answer Derivation Procedure and Architecture

LogAnswer consists of a multitude of interacting subsystems, see Figure 8.3. The easiest way to obtain a general understanding is by following the processing cycle starting with a question and ending with the answers.

Web-Based User Interface: The user enters a question into the LogAnswer user interface in the browser. Ideally this should be a properly formulated question with correct spelling and capitalization, terminated by a question mark, although LogAnswer has some robustness toward syntactic mistakes.

Deep Question Parsing: The WOCADI parser analyzes the question and creates its semantic representation in the MultiNet formalism. This processing

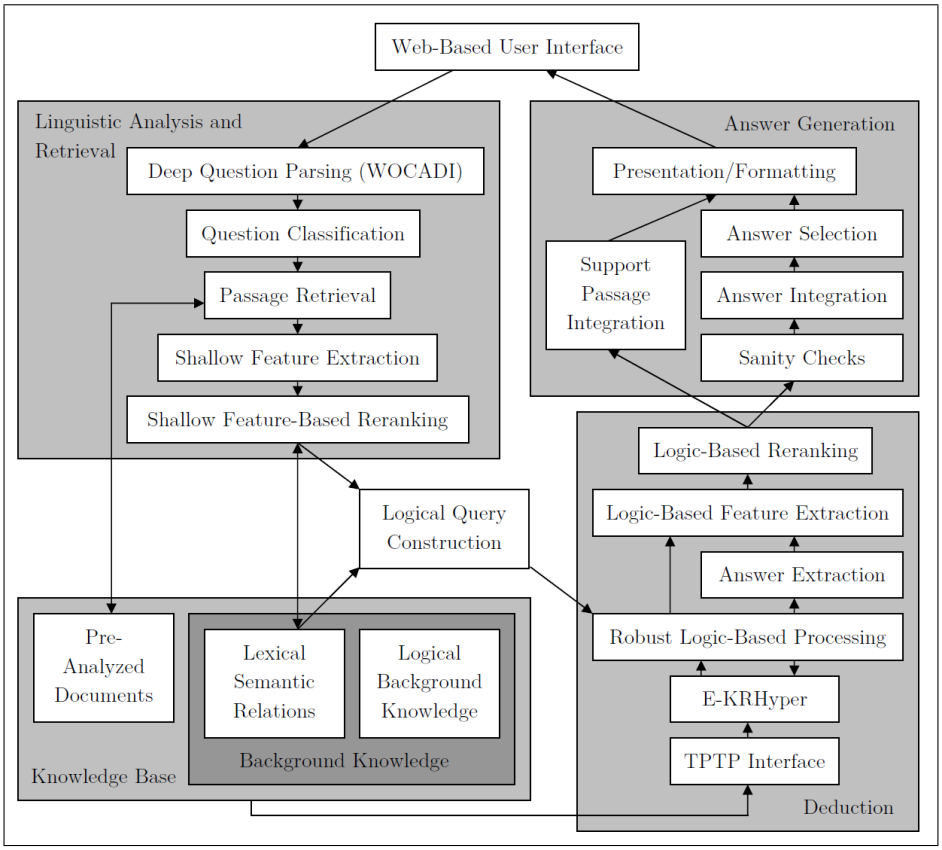


Figure 8.3: The architecture of LogAnswer

phase also involves a word sense disambiguation step using the HaGenLex lexicon. WOCADI also features a coreference resolution module which can handle follow-up questions involving pronouns and nominal anaphora.

Question Classification: The category of the question is determined: For example *factoid questions* (like *"What is the capital of Australia?"*) can be answered by logical means alone, whereas *definition questions* (like *"What is Australia?"*) necessitate filtering criteria that find knowledge which actually is defining, so that the answer will meet the requirement of completeness (see Section 4.2). Other categories cover questions regarding opinions, procedures, purposes and reasons. Another result of this phase is the *expected answer type*, distinguishing between questions asking for names of persons, locations, numbers and so on.

Passage Retrieval: This is an important preprocessing phase before the theorem prover can be employed. As mentioned before, the knowledge base is too large to be handled by an ATP system directly. It is based on 29 million sentences of a snapshot of the German Wikipedia. This data is stored locally for

faster access, and the snapshot is replaced periodically. Further source documents like the CLEF competition corpora can be added as well. The knowledge sources are not parsed anew for every question, instead all documents have been pre-analyzed by WOCADI, and their resulting MultiNet representations are indexed by their terms and and made accessible to the retrieval modules. Two such modules are available to LogAnswer, the aforementioned IRSAW and an alternative [9] based on Lucene.[4] Both operate with with sentence-sized passages, although the latter also uses descriptors resulting from coreference resolution of previous sentences. Up to 200 passages with MultiNet representations, the *answer candidates*, are retrieved from the knowledge base for each question. Apart from the processed documents the knowledge base also contains the *background knowledge*, consisting of a set of about 10,600 lexical-semantic facts which establish associations for example between certain nouns and verbs, and the logical background knowledge comprising nearly 200 logical rules based on the MultiNet formalism. No passages are extracted from the background knowledge, rather these general rules are used at different points during the processing.

Shallow Feature Extraction: The answer candidates may vary in quality and relevance, and in order to assess their importance a number of shallow linguistic features is computed for each passage, based on the earlier analysis of the question and the passage:

***failedMatch*:** the number of lexical concepts and numerals in the question which cannot be matched with the passage,

***matchRatio*:** the relative proportion of lexical concepts and numerals in the question which find a match in the passage,

***failedNames*:** the proper names occurring in the question, but not in the passage,

***containsBrackets*:** a boolean feature indicating whether the passage contains a pair of parentheses,[5]

eatFound: a boolean feature indicating that the passage contains words of the expected answer type,

***defLevel*:** a numeral feature indicating how useful the passage may be for definition questions, based on the presence of certain words and sentence structures,

***irScore*:** the passage score determined by the retrieval module in the previous phase.

Shallow Feature-Based Reranking: The passages are then ranked by their shallow features. This is handled by machine learning-based decision trees which aggregate the feature values into a probability for each passage, so that the deduction component can process the most promising answer candidates first,

[4] `http://lucene.apache.org`

[5] Parentheses often contain valuable factoid information, for example as in *"Canberra (Australia)"*.

a critical necessity given the short time limits and the fact that the passage retrieval only has a precision of about 3%.

Logical Query Construction: The question is translated into first-order logic, where it takes the form of a conjunction of positive literals with existentially quantified variables. These literals represent a MultiNet fragment where variables take the place of node identifiers. If the question asks for specific information, then this is represented by a special *FOCUS* variable. For example, *"Wer erfand Coca-Cola?"*[6] translates into the following logical query, with the *FOCUS* variable representing the person of interest:

$$\begin{aligned}\exists X_1 \exists X_2 \exists X_3 \exists FOCUS \quad & \mathrm{obj}(X_1, X_2) \wedge \mathrm{subs}(X_1, \textit{erfinden.1.1}) \\ & \wedge \mathrm{agt}(X_1, \textit{FOCUS}) \wedge \mathrm{attr}(X_2, X_3) \\ & \wedge \mathrm{sub}(X_3, \textit{name.1.1}) \wedge \mathrm{val}(X_3, \textit{coca-cola.0})\end{aligned}$$

This translation phase also normalizes all synonyms by replacing them with canonical representations specified by the set of lexical-semantic relations in the knowledge base.

To answer a query, its fragment must be matched against the MultiNet knowledge base, thereby instantiating the variables with concrete nodes. A proof attempt by the prover can then be regarded as the logical representation of this matching process: The proof is done by refutation, the query is treated as a negated conjecture, and a successful refutation means that the variables of the logical query have been instantiated by constants representing nodes in the MultiNet knowledge base. The clausal representation of the example question as used by E-KRHyper thus has the following form:

$$\begin{aligned}\{ & \neg\mathrm{obj}(X_1, X_2), \neg\mathrm{subs}(X_1, \textit{erfinden.1.1}), \neg\mathrm{agt}(X_1, \textit{FOCUS}), \\ & \neg\mathrm{attr}(X_2, X_3), \neg\mathrm{sub}(X_3, \textit{name.1.1}), \neg\mathrm{val}(X_3, \textit{coca-cola.0})\}\end{aligned}$$

The prover keeps track of the *FOCUS* variable throughout all clause transformations and inference steps, so that its binding can be extracted from a proof even if it has been renamed during the process.

An exception to the creation of purely conjunctive logical query representations occurs during the handling of disjunctive questions. The logical representation of a disjunctive question will contain disjunctive literals. An example of this is the following modification of the previous query: *"Wer erfand Coca-Cola oder Pepsi-Cola?"*[7]

This results in the following disjunctive logical query:

$$\begin{aligned}\exists X_1 \exists X_2 \exists X_3 \exists FOCUS \quad & \mathrm{obj}(X_1, X_2) \wedge \mathrm{subs}(X_1, \textit{erfinden.1.1}) \\ & \wedge \mathrm{agt}(X_1, \textit{FOCUS}) \wedge \mathrm{attr}(X_2, X_3) \\ & \wedge \mathrm{sub}(X_3, \textit{name.1.1}) \\ & \wedge (\mathrm{val}(X_3, \textit{coca-cola.0}) \vee \mathrm{val}(X_3, \textit{pepsi-cola.0}))\end{aligned}$$

[6] *"Who invented Coca-Cola?"*
[7] *"Who invented Coca-Cola or Pepsi-Cola?"*

After negation of this conjecture and the subsequent transformation into CNF, the logical query representation for E-KRHyper consists of multiple clauses as shown below:

$$\{\neg\text{obj}(X_1, X_2), \neg\text{subs}(X_1, \textit{erfinden.1.1}), \neg\text{agt}(X_1, \textit{FOCUS}),$$
$$\neg\text{attr}(X_2, X_3), \neg\text{sub}(X_3, \textit{name.1.1}), \neg\text{val}(X_3, \textit{coca-cola.0})\}$$
$$\{\neg\text{obj}(X_1, X_2), \neg\text{subs}(X_1, \textit{erfinden.1.1}), \neg\text{agt}(X_1, \textit{FOCUS}),$$
$$\neg\text{attr}(X_2, X_3), \neg\text{sub}(X_3, \textit{name.1.1}), \neg\text{val}(X_3, \textit{pepsi-cola.0})\}$$

Such clauses are handled simultaneously in a single proof attempt, and disproving one of them is sufficient for an answer. Without loss of generality we assume a single query clause in the sequel.

Robust Logic-Based Processing: For each of the ranked answer candidates the FOL representations of the query, the passage and the background knowledge are sent to the robust logic-based processing module which interacts with E-KRHyper. With the average translated passage consisting of 230 unit clauses and the background knowledge adding its 10,600 lexical-semantic relations and 200 logical rules, the prover has to operate on about 11,000 clauses in total for every answer candidate. This input size has remained fairly stable since the early version of LogAnswer [15], as the largest growth has been in the Wikipedia-based part of the knowledge base, resulting in more stored passages, whereas the background knowledge does not change much.

E-KRHyper constructs a tableau for this input set. In the current form the input clauses are always Horn clauses, so the tableau consists of a single branch. When the branch is closed, then E-KRHyper attempts to extract answer data from the proof, specifically from the final inference step. The term bound to the *FOCUS* variable in the final unifying substitution then represents the queried information.

Normally the query clause is the only negative clause and thus the only clause that can close a branch. However, E-KRHyper explicitly makes sure to accept only those proofs for answer extraction which finish with the query clause. This is because future knowledge base extensions may add negative clauses as constraints or contain other internal contradictions, and a branch that closes without instantiating the query clause does not yield an answer.

There is also the possibility of multiple query clauses. This can be due to a disjunctive query as described above, or because the sup-left or ref rules derive new query clauses from a query clause. In such cases it is sufficient for one of these clauses to close the branch, and only one answer will be extracted from this. Again E-KRHyper keeps track of the *FOCUS* variable throughout clause transformations and inference applications, so that an exact answer can also be extracted from derived query clauses.

A less clear situation can arise due to the presence of non-Horn clauses which cause branching in the hyper tableau. While this is impossible with the current knowledge base, further extensions could add such clauses. The recently proposed QA syntax addition to the TPTP allows this as well, although the issue of disjunctive answers is regarded as problematic.[8] The treatment of answers

[8] http://www.cs.miami.edu/~tptp/TPTP/Proposals/AnswerExtraction.html

in a tableau with branching largely depends on the intended semantics of the non-Horn clauses that cause the branching.

For example, consider the following pair of a question Q and a candidate passage C:

Q**:** *"Which airports serve Tokyo?"*

C**:** *"Visitors to Tokyo will land at Haneda Airport or at Narita Airport."*

Here the *"or"* could lead to two branches that each close with one of the airports as an answer, and together they could form a conjunctive answer *"Haneda Airport and Narita Airport"*. This does not work when the *"or"* is exclusive:

Q**:** *"What happens if I catch the bubonic plague?"*

C**:** *"Victims of the bubonic plague die within four days or survive disfigured."*

Again two answers are possible, but a combined conjunctive answer would be nonsensical (*"You die within four days and you survive disfigured."*). Instead the system should either present only one of the outcomes, or preferably show both as a combined disjunctive answer (*"You die within four days or you survive disfigured."*). The current behaviour of E-KRHyper is therefore to extract one answer from each branch that is closed by a query clause. It is then up to the overarching system to decide how to deal with multiple answers for one passage, by treating them as conjunctive or as disjunctive. Technically this makes it possible to obtain answers from incomplete proofs by considering only the branches closed so far.

The logic-based processing occurs under strict time limits, so it is possible that not all candidates for a question will be tested by the prover. This extends to individual proof attempts which are limited to a slice of the overall deduction time limit. These time slices play a role in ensuring robustness, which is done by *relaxation*: If E-KRHyper cannot find a proof for a passage within a specified time limit, then query literals are skipped until the remaining query can be proven. Relaxation is described in detail in Chapter 13. If no proof succeeds at all, E-KRHyper can also provide any partial proof substitutions which managed to instantiate some of the query literals.

Answer Extraction: If E-KRHyper manages to prove a question with respect to a candidate passage, then the answer binding from the proof is handed over to the main LogAnswer system. This data is used to extract the actual answer string from the original passage, a step that is necessary because the MultiNet node identifiers are often ill-suited as natural language responses, and an answer may require words associated with several nodes. Returning to the example in Figure 8.2, a successful proof for the query about the inventor of Coca-Cola would instantiate the *FOCUS*-variable with the node identifier *c31*. This node represents the person John Stith Pemberton. Based on this node the answer extraction will identify those nodes which store the given names and the surname and thereby construct the answer *"John Stith Pemberton"* from the passage.

Logic-Based Feature Extraction: Similar to the answer candidates the proofs may be of different quality, thanks to the relaxation which can skip literals and thereby make the query less specific. To determine the best proofs a number of features is extracted:

***skippedLitsLb*:** the number of literals skipped by relaxation,

***skippedLitsUb*:** the number of skipped literals plus the number of literals that remained unproven (partial proof),

***litRatioLb*:** the relative proportion of proved literals compared to all query literals,

***litRatioUb*:** the relative proportion of unskipped literals compared to all query literals,

***boundFocus*:** a boolean feature indicating whether the *FOCUS* variable was bound,

***npFocus*:** a boolean feature indicating whether the *FOCUS* variable was bound to a node corresponding to a nominal phrase,

***focusEatMatch*:** a boolean feature indicating whether expected answer type matches the answer type found in the passage,

***focusDefLevel*:** analogous to the shallow *defLevel* feature, this feature indicates the relevance of the answer binding for definition questions.

Logic-Based Reranking: Analogous to the shallow feature-based reranking, the passages and the answers are ranked by a confidence score computed by decision trees which now utilize both the shallow and the logic-based features. Depending on the use case it is possible to apply a filter here by introducing an answer acceptance threshold θ, so that any answer with a score below θ is discarded for being too uncertain. This is useful in situations when incorrect answers will be perceived as disturbing. As the logical proof has a great influence on the score of an answer, the proof quality can become a deciding factor in the decision whether to keep or reject an answer. The optimum value for this threshold is highly dependent on the complexity of the questions, their syntactic correctness, the subject matter and so on, so if a θ is to be used, it must be computed for the specific scenario by machine learning on ample training data.

Support Passage Selection: Depending on the precise use case, LogAnswer may only have to produce passages instead of exact answers; this has been the case in some CLEF competitions. In such circumstances the (usually five) top-ranked passages are selected for presentation. The shallow retrieval methods in combination with the large knowledge base normally ensure that there are indeed at least five such passages at this stage, even if their confidence scores may be very low. In rare cases though it can happen that there are less passages.

Sanity Checks: For exact answers on the other hand there is some additional filtering. This prevents presenting the same answer twice (although multiple occurrences of the same answer do increase its relevance ranking). Trivial answers which merely repeat question terms are discarded as well, for example: *"What is a computer?"* - *"a computer"*. The best five answers are selected for presentation to the user. Again this number can be less if not enough answers were found or passed the filtering stages.

Apart from answering questions, one goal of the described arrangement of components and their interaction in the processing cycle is that LogAnswer can function as an anytime algorithm. The more time LogAnswer is allowed for the processing of a question, the more answer candidates can be tested in E-KRHyper, and the more inferences can be computed in each proof attempt. Even short time limits are sufficient for finding some answers, and the number and quality of the answers improve when the limits are increased. This means that LogAnswer can be scaled up and down to different usage scenarios without having to implement major changes to the inner workings of the system, making it flexible enough to serve both as a quickly responding search engine replacement on the web and as a QA system with thorough and deep reasoning, for example as a research tool or in QA competitions.

This concludes the high-level description of LogAnswer. The following chapters will address specific issues in adapting the theorem prover E-KRHyper to this QA system.

Chapter 9

Suitability of ATP Strategies

Before delving into detailed adaptations of an automated theorem prover to specific QA issues it makes sense to consider whether there are some general features that make one prover more suitable to question answering than another. Obviously some provers may have advantages towards embedding by having more open interfaces, supporting more input syntaxes and so on, but that is not the concern of this chapter. Rather, we want to investigate whether there are design decisions in the development of a prover's proof procedure that provide a benefit when processing reasoning problems stemming from QA tasks.

The chapter is structured as follows. The first section gives a high-level system description of the theorem prover E-Darwin which I developed and used during the LogAnswer project. This includes a short excursus about ATP debugging, which gives reasons for having a second theorem prover within the project. The third section then explains how the differences between E-KRHyper and E-Darwin lead to significant differences in their performance on LogAnswer reasoning problems, and a short evaluation shows that this applies to automated theorem provers in general.

9.1 The Theorem Prover E-Darwin

E-Darwin [3] is an automated theorem prover implementing the model evolution calculus [BT03] and several of its variations. Within the LogAnswer project it served as a secondary prover that helped during the debugging and benchmarking of E-KRHyper. It has not been embedded in LogAnswer, and therefore its operation will only be discussed in the broadest strokes, as any deeper level of detail is not necessary for the purpose of this chapter.

9.1.1 Background and Development History

The model evolution calculus was originally implemented by Alexander Fuchs in the theorem prover *Darwin* [BFT06]. Later versions of this calculus added different ways of equality reasoning [BT05][3]. E-Darwin is intended as a testbed for these variants of model evolution, and I implemented E-Darwin as a fork of

Darwin. The development of E-Darwin and its relation to Darwin thus mirror that of E-KRHyper and KRHyper. However, while the KRHyper-provers emphasize embedding in knowledge representation applications and use techniques from deductive databases, the Darwin-systems are rather intended as stand-alone provers, and they rely more strongly on standard ATP design principles.

9.1.2 Usage Information

E-Darwin is implemented in OCaml, the first of many similarities to E-KRHyper. Likewise, E-Darwin supports input in TPTP and PROTEIN syntax. The system operates on clauses in clause normal form. The Darwin-provers have traditionally clausified input formulas using the prover E [Sch02], although as of version 1.4 E-Darwin has adopted the clausification module developed for E-KRHyper, and it allows the user to select which clausifier to apply. Results can be returned in various forms, including the SZS-compliant result status. For satisfiable input E-Darwin can return a model if it terminates. Proofs are provided as traces, i.e. in the form of listings of the derivation steps taken, with the level of detail being selectable. The current E-Darwin 1.5 is available under the GNU General Public License at the E-Darwin website.[1]

9.1.3 Proof Procedure

The model evolution calculus with equality operates on *sequents*, and a sequent consists of a set of clauses and a *context*, the latter being a set of literals that can be regarded as a model under construction. Each inference rule derives a new sequent with a modified clause set or context. Rules based on superposition can use equations for rewriting. A splitting rule derives two disjunctive sequents which must be considered one by one. Simplification and subsumption rules allow the calculus to exploit a range of redundancy criteria. If a sequent is free of contradictions yet has no non-redundant inference possibilities, then its context represents a model. If only contradictory contexts are derived, then the input is unsatisfiable.

From an implementational point of view a sequent is roughly analogous to a branch in an E-hyper tableau. Similar to E-KRHyper, the proof procedure of E-Darwin can be divided into an upper level and a lower level algorithm, see the pseudo-code Algorithms 9.1 and 9.2. The upper level algorithm handles splitting and backtracking, while the lower level evaluates sequents in a linear manner between splits. The upper level is almost identical to the one in E-KRHyper (see Algorithm 7.5): Both systems postpone splitting as long as possible, both can output a model when a branch (sequent) in the derivation tree is exhausted, and both backtrack when a branch (sequent) is closed. The only significant difference here is that E-Darwin has no option to retain results above the weight limit.

The lower level sets the systems apart, as E-Darwin does not use semi-naive evaluation. In order to minimize clutter in the pseudo-code the sequent is always represented just as the *sequent* variable, and when adding elements to the sequent or using the sequent in some other way, it is assumed that the proper subset of the sequent is used, i.e. either the set of clauses or the context. The function EVALUATE_SEQUENT takes as its argument *new* a set of clauses and

[1] `http://userpages.uni-koblenz.de/~bpelzer/edarwin`

Algorithm 9.1 At the upper level the E-Darwin split control loop handles splitting and backtracking.

```
function EVALUATE_CHOICE_POINT(choicePoint)
    if choicePoint = root then
        sequentResult := EVALUATE_SEQUENT(input reasoning problem)
    else
        sequentResult := EVALUATE_SEQUENT({split literal of choicePoint})
    end if
    if sequentResult = closed then
        if unprocessedSplitStack ≠ {} then
            nextChoicePoint := pop(unprocessedSplitStack);
            sequent := backtrack(sequent, nextChoicePoint);
            candidates := backtrack(candidates, nextChoicePoint);
            splitCandidates :=
                  backtrack(splitCandidates, nextChoicePoint);
            EVALUATE_CHOICE_POINT(nextChoicePoint)
        else
            return closed
        end if
    else
        if splitCandidates ≠ {} then
            split := take a split from splitCandidates;
            push(literals of split, unprocessedSplitStack);
            nextChoicePoint := pop(unprocessedSplitStack);
            EVALUATE_CHOICE_POINT(nextChoicePoint)
        else
            if weightLimit has been exceeded then
                increase(weightLimit);
                nextChoicePoint := choice point of first
                                       weightLimit transgression;
                sequent := backtrack(sequent, nextChoicePoint);
                candidates := backtrack(candidates, nextChoicePoint);
                splitCandidates :=
                      backtrack(splitCandidates, nextChoicePoint);
                EVALUATE_CHOICE_POINT(nextChoicePoint)
            else
                return exhausted
            end if
        end if
    end if
end function
```

literals. In the initial call *new* is the clause set formed by the input reasoning problem. Later on this function is called after each split, and *new* then only consists of the respective split literal. *new* is simplified by the *sequent* in a reduction phase, then added to the *sequent*, and finally used to simplify the *sequent* as well. In the initial call this means for example that input clauses may subsume each other. After that the actual reasoning begins. E-Darwin

Algorithm 9.2 The lower level in E-Darwin evaluates sequents between splits.

```
function EVALUATE_SEQUENT(new)
    // In the first call, new is the input clause set.
    // In subsequent calls it is a selected split literal.
    new := reduce new by sequent;
    sequent := sequent ∪ new;
    sequent := reduce sequent by new;
    conclusions := all inferences with new,
                   using other premises from sequent;
    candidates := candidates ∪ (non-splits from conclusions);
    splitCandidates := splitCandidates ∪ (splits from conclusions);
    while (candidates ≠ {}) ∧ (contradiction not found) do
        selected := select one from candidates;
        candidates := candidates \ selected;
        selected := reduce selected by sequent;
        sequent := sequent ∪{selected};
        sequent := reduce sequent by selected;
        conclusions := all inferences with selected,
                       using other premises from sequent;
        candidates := candidates ∪ (non-splits from conclusions);
        splitCandidates := splitCandidates ∪ (splits from conclusions)
    end while
    if contradiction found then
        return closed
    else
        return exhausted
    end if
end function
```

exhausts all inference possibilities that involve at least one element of *new* as a premise, taking other premises from the *sequent*. The results that remain within the *weightLimit*, temporarily stored in *conclusions*, are then put either into *candidates*, or into *splitCandidates* in the case of disjunctive results.

A loop then proceeds as follows: In each iteration one of the *candidates* is selected heuristically. Factors influencing this selection are size and weight measures, age, and an estimate of the likelihood that the selected candidate can lead to a contradiction. Naturally, the latter criterion is so important that numerous lookahead functions continuously monitor the *candidates* in case changes to the sequent make a candidate contradictory. Once a candidate has become *selected*, its processing mirrors that of *new*. It is simplified by the *sequent*, added to the *sequent* and then in turn may simplify the *sequent*. All inference possibilities with the *selected* candidate are exhausted, and the results are stored in the respective candidate sets. The loop continues until there are no non-splitting *candidates* or a contradiction has been found. At that point the upper level algorithm takes over again.

9.1.4 Implementation Details

E-Darwin and E-KRHyper have many things in common due to their heritage. The original Darwin used discrimination tree indexing code adapted from the original KRHyper. The indexing of multi-literal clauses in E-KRHyper (see Chapter 10) builds upon this, and we first implemented and tested it in E-Darwin before porting it to E-KRHyper. Darwin came with a parser for clause normal form input in TPTP syntax; we ported this to E-KRHyper, where we extended it with the ability to process TPTP input in formula form by equipping it with a dedicated clausifier, and then this enhanced parser was ported back to E-Darwin.

However, some differences should be mentioned. E-Darwin has no layered indexing. When backtracking to a different derivation branch, invalidated index entries must be removed from the discrimination trees one by one, and branches left without entries are pruned.

Another difference lies in the internal representation of terms. Unlike E-KRHyper, terms in E-Darwin are *shared* as much as possible. This means that all occurrences of a given term or subterm within a set of clauses are actually represented by pointers to a single term. This is advantageous in that memory is saved, and tests for syntactic term equality, a frequent operation in indexing, unification and so on, can often be done just by verifying pointer identity, rather than having to compare terms subterm by subterm. There are also drawbacks: To ensure that no term is stored more than once, whenever a new term is formed, for example through the application of a substitution, it has to be matched against a term database to find a possible earlier representation of that term. During this initial lookup the term comparison cannot yet rely on pointer identity, so every such term creation requires one costly database index search. Shared terms also mean problems for rewriting: If a subterm s of a term t within a literal of a clause is to be rewritten by a term r (i.e. $t[s]$ is about to become $t[r]$), then care must be taken to ensure that other occurrences of t, whether within the same clause or in others, are not accidentally rewritten through a destructive modification of the one canonical representation of t. This can be solved by creating and using temporary copies of terms that are about to be rewritten, but this adds a substantial overhead to operations that occur very frequently during the runtime of a prover.[2]

The usage of term sharing is a fundamental design decision with pervasive consequences for the implementation of a prover, so it is not easy to convert a prover from shared to non-shared terms or vice versa. As the benefit situation is unclear and experiments with partial term sharing in E-KRHyper were inconclusive, we left both E-Darwin and E-KRHyper with their respective originally chosen term representation method.

9.1.5 Evaluation

E-Darwin generally outperforms E-KRHyper. We tested E-Darwin on the 15,550 FOL problems of the TPTP v5.3.0. An Intel Q9950 CPU with 2.83 GHz was used, and the memory usage was limited to 1 GB of RAM, while the time limit per problem was set to 300 seconds. Under these conditions E-Darwin solved

[2]Harald Ganzinger and Robert Nieuwenhuis estimate that theorem provers spend about 90% of the time on demodulation [Nie99], in other words on rewrite operations.

6,467 problems, corresponding to 41.6% of the test set, or 33.3% of the total TPTP with 19,446 problems. The hardest problem solved was *LAT298+1* with a rating of 1.0, indicating that no other current SOTAC prover solves this problem. When E-Darwin was introduced [3], it was tested on the TPTP v4.0.1 and solved six problems with a 1.0 rating at that time, namely *ALG035+1*, *GRP197-1*, *NUM378+1.020.015*, *PRO016+1*, *SWV527-1.040* and *SWV527-1.050*. The current E-Darwin still solves these problems, although they have since been downrated in their difficulty.

In the official TPTP test result listings[3] E-Darwin is listed in its 1.4 version from 2011, in which it solved 36% of the tested 15,550 FOL problems or 28% of the total TPTP. This version featured an inefficient experimental inference implementation which has since been improved, explaining the difference to our own results with the current E-Darwin 1.5. Nevertheless within these official listings E-Darwin is the 11th best prover out of the 62 tested on the TPTP v5.3.0, or the 9th best when accounting for some provers sharing positions due to showing the same performance. All else remaining equal E-Darwin 1.5 could expect to rise to the 7th position in this listing.

E-Darwin has participated in CASC since 2008. As it serves as a testbed for the model evolution calculus, the implementation changes from year to year are not so much performance improvements as they are experiments for calculus modifications. Therefore it is inappropriate to identify a general trend in E-Darwin's performance over the years. However, usually E-Darwin has outperformed E-KRHyper in most categories, with the exception of 2011 where the aforementioned troublesome 1.4 version participated. E-Darwin is more of a generalist than the original Darwin. Darwin won the EPR division (effectively propositional problems) both in 2006 and 2007, yet at the same time failed the Otter benchmark in numerous categories. E-Darwin cannot quite match the EPR performance of Darwin, but on the other hand, in the less problematic 2010 participation E-Darwin surpassed Otter in several divisions and categories where Darwin used to be inferior, including CNF, FNE (first-order formula problems without equality), HNE (Horn without equality), NEQ (non-Horn with equality) and PEQ (purely equational problems).

9.1.6 Excursus: ATP Debugging

One might question the value of maintaining a second ATP system, in particular one as intricate as E-Darwin. There are two answers to this. Firstly, the effort is not as large as it may seem, because as mentioned in Section 9.1.4 E-Darwin and E-KRHyper share several complex modules. Also, as E-Darwin focuses on traditional FOL theorem proving, its implementation neither has to deal with logic extensions beyond equality nor with features for embedding. Secondly, an alternative prover with a readily available, accessible and well understood source code is highly useful during debugging.

Very little is written about the debugging of automated theorem provers, which is surprising given the amount of time an ATP developer has to spend testing and debugging a prover. Bugs in ATP systems can be grouped into *soundness bugs* and *miscellaneous bugs*. The latter cover a wide range of programming errors that manifest in the form of crashes, error messages or other

[3]Most recent result listing: `http://www.cs.miami.edu/~tptp/TPTP/Results.html`
Our data is based on the results retrieved on 1 May 2012.

faulty output; their repair is usually straightforward and such bugs are thus just a minor nuisance of little interest. Soundness bugs on the other hand are vastly more difficult to handle. They reveal themselves through testing, when the prover decides a problem with a result contrary to the known correct result. That is, the prover claims a satisfiable problem to be unsatisfiable or an unsatisfiable problem to be satisfiable. Technically it might be a misnomer to refer to the latter case as a soundness bug rather than a completeness bug, but this loose usage of "soundness" when referring to bugs is commonplace in the ATP community and CASC. Besides, the very same bug often destroys both soundness and completeness. For example, if a faulty inference implementation derives a non-consequence clause C' instead of C, then the system is unsound because it derives a non-consequence, and it is incomplete because it omits deriving the consequence C. Thus one bug can cause both *false satisfiability* and *false unsatisfiability* results, so in the following we will use these two notions, as they are the starting points of discovering and debugging a soundness bug and for each of them there are methods that are more effective.

Regardless of whether we are dealing with false satisfiability or false unsatisfiability, there is often the problem that the undesired result occurs after several minutes of computation, during which thousands, maybe even millions of inferences may have taken place. Therefore it is often not obvious where exactly the prover made a mistake. Intuitively it may then make sense to verify the erroneous proof or model, but this is difficult in practice. Proof output of provers often has a proprietary syntax, and of course the proofs use the inference rules of the specific implemented calculus, so they are unlikely to be compatible with existing automated proof checking programs. When producing condensed proofs (without unnecessary inference steps) like E-KRHyper does, then there is also the motivation to compress the information about inferences as much as possible, as the prover has to keep and accumulate this data throughout the derivation in order to show only the relevant inference steps at the end. This means that individual derivation steps in the proof output can be difficult to verify. For example, E-KRHyper compounds multiple consecutive simplification operations on a clause into a single step in the proof, listing only the starting clause, the final clause, and the equation units involved. Listing the intermediate results and the exact rewrite positions could clarify such steps, but this would make a memory-hungry prover consume even more memory. False models are even less helpful: Not only do they have the same problem of compatibility to third party tools, but also even if some automated tool could confirm that a false model truly is no model, then this would not tell where the prover went wrong or what inference step is missing.

Fault Condensation:

A helpful first step is to condense the reasoning problem to the smallest subset of its clauses that still induces the faulty behaviour in the prover. The idea is to shorten the derivation time until the erroneous result occurs, thus speeding up the subsequent debugging process, because having to wait for several minutes between each minor adjustment and test run is tedious. For false unsatisfiability results this is easily achieved by deleting one clause after another while testing to ensure that the shrinking subset remains "unsatisfiable". E-KRHyper has a function specifically for this purpose which lists the input clauses involved in

an unsatisfiability proof in a machine-readable form. That way a false proof can be turned into a shortened input set without having to reduce and test the original reasoning problem iteratively. When applying condensation to an unsatisfiable problem with a false satisfiability result there is the impediment that the subset of an unsatisifable clause set may actually be satisfiable, so by simply deleting clauses one may end up turning a false satisfiability into a true satisfiability result, which is not helpful. Here a second prover is useful: Whenever the reasoning problem has been reduced by deleting clauses, both the faulty prover and the second prover test the reduced set. If the second prover finds this input to be unsatisfiable and the faulty prover still produces the false satisfiability result, then the reduction was successful, and further condensing may be possible. On the other hand, once both provers show satisfiability, then the condensing has gone too far and it is no longer certain that the reduced input still triggers the bug in the faulty prover.

Branching Reduction

The next step is to further condense the faulty derivation by a reduction of branching. In the case of a false satisfiability result the final open branch is likely to be involved in the bug, since all branches should have been closed. Any previously closed branches are less interesting. To speed up debugging it can thus be useful to skip these earlier branches by forcing the faulty prover onto the erroneously open branch right away. This can be achieved by obtaining the "model" M represented by the branch and adding the units to the input problem one by one, effectively turning them into input axioms. Of particular interest here are those units which were derived by splitting, since their presence will then prevent the prover from exploring the other branches resulting from those splits.

For a false unsatisfiability result a reduction of branching faces the problem that even satisfiable input may lead to many closed branches, and often it is not known which particular branch was closed in error. In the rare cases where this branch is known, the faulty proof provides information about which splittings led to the specific branch, and the respective units can then be added to the input problem in order to force the prover onto that branch as explained above. When the faulty branch is not known, eliminating branches at random bears the risk of accidentally eliminating the faulty branch, thereby actually rendering the clause set truly unsatisfiable. Here it can be helpful to proceed iteratively: From the faulty proof one split unit is selected and added to the input problem in order to eliminate one splitting. Then the modified input is tested both with the faulty prover and the second prover. If the faulty prover repeats the unsatisfiability result while the second prover claims the opposite, then the reduction was useful, as the condensed problem still triggers the bug. However, if both provers agree on the input being unsatisfiable, then the reduction turned the input truly unsatisfiable. Likewise, if both provers agree on the modified input being satisfiable, then it probably is satisfiable, but it no longer triggers the bug in the faulty prover. Thus, whenever both provers agree, then that particular branch reduction should be reversed, as it is useless for debugging.

When there is a large number of splittings in a faulty refutation, then eliminating them one by one may become tedious. A more drastic measure then is to delete positive literals from non-Horn clauses, i.e. by selecting a clause

$C = A, \mathcal{A} \leftarrow \mathcal{B}$ and replacing it with $C' = \mathcal{A} \leftarrow \mathcal{B}$. That way multiple splittings can be prevented by one modification. As above, the modified input must be tested by a second prover to ensure that the satisfiability status has not been affected.

It is not always possible to reduce a faulty derivation to one branch. For example, the erroneously open branch of a false satisfiability result may become closed when computed on its own. In such cases the bug is often found in the splitting or backtracking implementations, in such manner that clauses from previously closed branches are carried over into the faulty branch by accident, or clauses made redundant in an earlier branch are not reintroduced properly during backtracking.

Inference Sequence Testing:

When a flawed branch has been isolated successfully, the derivation is often so short that the error becomes obvious. However, when the branch is long, it may be useful to verify the derivation automatically, which is easier now that no branching is involved. A faulty refutational proof can be broken down into a sequence of inference steps, each of the form $\mathcal{C} \Rightarrow \mathcal{D}$ with premises $\mathcal{C}$ and conclusions $\mathcal{D}$. Each such inference step can be turned into a compact reasoning problem $\mathcal{C} \wedge \neg\mathcal{D}$. These individual problems are usually sufficiently simple to be almost instantly proven by a second prover, even if they involve multiple simplification operations as mentioned above. E-KRHyper is equipped with functions and scripts that can automatically test a branch like this using E-Darwin.

When dealing with a false satisfiability result, then there is no refutation to sequentialize. In such cases the proofs for individual units of the faulty model may be scrutinized in a similar manner. Often though the bug is then found in the redundancy handling, with non-redundant clauses being deactivated by accident. Switching off the redundancy handling, either entirely or for example by iteratively disabling the first or last n clause deactivations, can help in discovering which clause should not have been removed from the reasoning.

Example Bug:

For an example of a typical bug, consider this set of clauses:

C_1**:** $\leftarrow cA(x), cB(x)$

C_2**:** $cB(x) \leftarrow cB(y), x \simeq y$

C_3**:** $ia \simeq ib \leftarrow$

C_4**:** $cA(ia) \leftarrow$

C_5**:** $cB(ib) \leftarrow$

The clauses were extracted from the TPTP problem *KRS163+1*, normally an unsatisfiable problem with 19 clauses in the CNF representation, some of them non-Horn. E-KRHyper claimed to find a model for the original problem after approximately 70 seconds of computation. The problem could be condensed to the five clauses above; together they remain unsatisfiable, yet E-KRHyper

claimed to find a model almost instantly. It should be possible to derive a refutation in two steps:

1. Derive $C_6 = cB(ia) \leftarrow$ from C_3 and C_5 using unit-sup-right.
2. Close the branch using C_1, C_4 and C_6.

Notably, C_2 is absent from the proof, and technically the clause is tautological, but it was critical for the bug. Recall that E-KRHyper operates in rounds, performing several inferences in each round and temporarily storing the results in a set of *conclusions*, from where they will be moved to the tableau after the round. In the first round E-KRHyper initially derived C_6, but it did so using hyper extension with the selected clause C_2 and the units/literals C_3 and C_5. In the same round the prover then also derived $C_7 = cB(ia) \leftarrow$ from C_3 and C_5 using unit-sup-right, in other words the expected first step above. This inference made C_5 redundant, so it was deactivated. As C_7 was being entered into the *conclusions*, it was recognized to be subsumed by the already present C_6 and therefore discarded. At the end of the round C_6 should then be inserted into the tableau, but a final check recognized that one of its premises, namely C_5, had become redundant due to another inference during the course of the same round. This made the whole inference deriving C_6 redundant as well, so C_6 was likewise discarded. With both instances of $cB(ia) \leftarrow$ discarded, the closing inference was then no longer possible.

The problem here was that the *conclusions* set utilized two optimizations which together destroyed the completeness. By both testing for subsumption between new conclusions and also rejecting inference results whose premises had become redundant by other inferences during the same round, it effectively allowed two instances of the same clause to discard each other, leaving none of them. As the internal subsumption test between fresh conclusions was only a minor optimization, we chose to disable it in order to restore the completeness of the implementation.

Debugging an automated theorem prover can involve a considerable amount of work, with single faults sometimes requiring several days of constant testing and investigation. A second prover is a valuable tool for verification during this work, and if it is as closely related to the main prover as E-Darwin is to E-KRHyper, then it can sometimes be employed as a test-bed for specific components of the main prover by integrating them temporarily.

9.2 ATP Strategies and LogAnswer Problems

The evaluation in Section 9.1.5 shows that E-Darwin outperforms E-KRHyper when it comes to theorem proving in general, on reasoning problems from the TPTP. With both provers at our disposal, it is reasonable to consider embedding E-Darwin into LogAnswer instead of E-KRHyper. However, early testing with several provers showed that good general performance does not necessarily carry over to the reasoning problems expected in LogAnswer [8]. We used a test set[4]

[4]The set is available online:
`http://www.loganswer.de/resources/loganswer_tptp_problems.tar.gz`

obtained from the participation of a predecessor system to LogAnswer in the German language category of the CLEF 2007 competition track for QA systems (QA@CLEF 2007) [GFH+07]. The set consists of 1,805 problems, all of which are Horn problems without equality, but with an infinite Herbrand universe due to function symbols. The problems are very similar in size and structure to the expected LogAnswer problems (see Section 8.4), as they have been generated by the same methods, only from slightly older knowledge bases. On average the problems have 10,590 clauses, ranging from the smallest problem with 10,360 to the largest with 12,077. Most clauses are unit clauses, with an average of 300 multi-literal clauses per problem. While all of the problems are solvable in their original MultiNet form, the incomplete translation method into FOL does not guarantee that all problems in our test set are theorems, although most of them have been shown to be solvable by now.

We did not include the E-Darwin test results in [8] because of soundness problems with the system at the time, but the testing indicated that compared to E-KRHyper, E-Darwin would require several times as much time on average to solve a problem, and it would solve less problems than E-KRHyper even under generous time limits. For this dissertation we have tested the same problem set with current ATP versions including a corrected E-Darwin.

However, let us first consider the reasons for the discrepance in performance between E-KRHyper and E-Darwin, since this difference caused us to investigate the matter in the first place. Both provers are strongly related and share several components, including the clausifier and the indexing. Both use very similar upper level algorithms, see Algorithms 7.5 and 9.1. Even their calculi are in some ways related. On problems like those in the test set, E-KRHyper operates almost exclusively with the hyper extension rule, while E-Darwin does the same with its *Assert*-rule - both are effectively identical when it comes to Horn clauses. They do have a major difference in their lower level algorithms, though: While E-KRHyper uses the strategy based on semi-naive evaluation, the corresponding loop in E-Darwin is a variant of the *given-clause algorithm*. This algorithm was originally devised as an implementation of the *set of support strategy* [WOLB84] and introduced by the prover Otter [McC03], which is why it is often simply referred to as the *Otter-loop*. In a similar vein the specific variant of the given-clause algorithm in E-Darwin is normally called a *DISCOUNT-loop* due to its introduction by the *DISCOUNT* prover [DKS96]. The Otter-loop differs from the DISCOUNT-loop only in that the former also involves the set of *candidates* (usually referred to as the *set of support*) in simplification and subsumption operations. This is often regarded as causing more effort than gains, and the DISCOUNT-loop seeks to remedy this by keeping the set of support more passive.

These differences do not matter in the context of this chapter, and for our purposes we here omit all reduction phases to obtain the basic given-clause algorithm as shown in the pseudo-code of Algorithm 9.3. For a comparison Algorithm 9.4 shows a similarly distilled and simplified representation of the strategy in E-KRHyper based on semi-naive evaluation. To highlight their differences and commonalities, both representations adopt the standard terminology of *usable* for the set of clauses that can participate in inferences and *set*

A selection of these problems has since become part of the TPTP, where they are filed as *CSR112+1* to *CSR116+47*.

Algorithm 9.3 The basic given-clause algorithm

```
set of support := input reasoning problem;
usable := {};
while (set of support ≠ {})∧(refutation not found) do
    given clause := select and remove clause from set of support;
    usable := usable ∪{given clause};
    conclusions := all inferences with given clause,
                   using other premises from usable;
    set of support := set of support ∪ conclusions;
end while
if refutation found then
    return refutation
else
    give up
end if
```

Algorithm 9.4 The core E-KRHyper strategy based on semi-naive evaluation

```
usable := input reasoning problem;
conclusions := all inferences with usable;
while (conclusions ≠ {})∧(refutation not found) do
    previous conclusions := conclusions;
    conclusions := all inferences with previous conclusions,
                   using other premises from usable;
    usable := usable ∪ previous conclusions;
end while
if refutation found then
    return refutation
else
    return model
end if
```

of support for the more passive candidate clauses in the given-clause algorithm. Both also omit any references to splitting, as this is mostly handled outside of these algorithms.

The critical difference between the loops lies in the amount of clauses used for inferences during each loop iteration. The given-clause algorithm selects only one clause from the set of support, computes all inferences involving that clause, and then adds the conclusions to the set of support. In the semi-naive loop the set of conclusions has a function similar to the set of support in the way it stores the inference results. However, here all these conclusion clauses are selected for inferencing in the next loop iteration.

The given-clause algorithm is the predominant strategy in FOL theorem proving. Of the FOL provers participating in the CASC of 2010 [Sut11] and 2011 [Sut12], at least 80%[5] use some variation of the given-clause algorithm. This includes top-performing systems like Vampire and E.

[5]The percentage is a conservative estimate, and some of the more exotic systems could be generalized as also using the given-clause algorithm, but doing so might be an oversimplification that does not do justice to these implementations.

However, when dealing with reasoning problems as they occur in LogAnswer the given-clause algorithm may be at a disadvantage compared to the strategy implemented in E-KRHyper. This can be explained via the notion of *proof depth*:

Definition 9.1 (Proof Depth)**.** *As a preliminary notion, let the* inference depth *be a mapping of clauses to natural numbers. Given a clause set $\mathcal{C}$ and a set of inference rules (a calculus) $\mathcal{R}$, the inference depth of a clause C is determined recursively:*

- *If C is an input clause ($C \in \mathcal{C}$), then its inference depth is* 0*.*
- *If C is derived by applying an inference rule $R \in \mathcal{R}$ to a set of premise clauses $\mathcal{P}$ ($\mathcal{P} \Rightarrow_R C$), then the inference depth of C is equal to* 1+ *the maximum inference depth of any clause in $\mathcal{P}$.*

Assume $\mathcal{C}$ to be unsatisfiable and let P be a proof for this unsatisfiability using the calculus $\mathcal{R}$, then the proof depth *of P is the maximum inference depth of any clause used in P.*

Regarding the inference depth, note that we are dealing with specific clause occurrences in a derivation, so it is possible to have multiple clauses $C, C', \ldots$ which are variants and which each have their own (possibly different) inference depth, depending on how that particular variant was derived.

As the proof depth is determined by a particular proof using a specific calculus, a given problem may have many different associated proof depths. Also, when a proof requires branching, then this measure alone is insufficient to obtain any good idea of the complexity of a proof, as even a shallow proof may contain excessive branching. Despite these limits the proof depth should be a useful measure in our particular case of evaluating provers for LogAnswer. The test set is Horn, which is generally favorable towards avoiding disjunctive splitting. Indeed, our primary test candidates E-KRHyper and E-Darwin both do not need to split on these problems. In such a non-branching proof the proof depth is identical to the proof length, i.e. the total number of inference steps, while in a branching proof it is equal to the length of the longest branch.

Now let us assume some calculus $\mathcal{R}$ both in an implementation $\mathcal{R}^{GCA}$ using the given-clause algorithm and an implementation $\mathcal{R}^{SNE}$ using the algorithm based on semi-naive evaluation. If a Horn-problem $\mathcal{C}$ can be solved with a proof depth n, then $\mathcal{R}^{GCA}$ can find the proof in at least n loop iterations, while $\mathcal{R}^{SNE}$ is guaranteed to find it in n loop iterations. This is not necessarily an advantage for semi-naive evaluation, as each iteration has to perform more inference work. Assuming that the average selected premise clause (i.e. the *given clause* in $\mathcal{R}^{GCA}$ and one of the clauses in *previous conclusions* in $\mathcal{R}^{SNE}$) results in m new conclusion clauses, $\mathcal{R}^{SNE}$ sees an exponential growth in derived clauses, with $\sum_{i=1}^{n} m^i$ clauses having been derived by the n-th round, whereas $\mathcal{R}^{GCA}$ will only derive $m \cdot n$ clauses. In an ideal situation where only minimal inferencing is possible and the proof is linear, both algorithms will find the proof after the same amount of inference applications. In practice, though, $\mathcal{R}^{SNE}$ is often likely to become overwhelmed by excessive clauses in later iterations.

However, this does not apply to QA-related reasoning problems as they occur in LogAnswer. Drawing from the upcoming evaluation of the QA problem set with E-KRHyper, there are proofs for at least 1,728 of the 1,805 problems

(95.7%). Among these the average hyper tableaux proof depth is 2.125, ranging from 1 for the most shallow proofs to 5 for the deepest ones. The average proof is based on 10.8 input clauses, ranging from 2 to 35. This means that the average solved problem can be proven using less than 0.1% of its clauses, and on average over 99.9% of the input clauses turn out to have been essentially useless baggage for the theorem prover. An ATP system working on such problems must therefore be able to find a relatively compact proof among mostly irrelevant inference possibilities. This irrelevance is not detected by the usual redundancy criteria employed by reasoning calculi, such as subsumption and tautology elimination.

When dealing with LogAnswer problems the behaviour of the semi-naive evaluation-based $\mathcal{R}^{SNE}$ gives it an advantage over the given clause algorithm $\mathcal{R}^{GCA}$ due to the properties of the search space with respect to finding a proof. Let us represent this search space by arranging the clauses of a derivation in a directed graph $G = (N, A)$, where the nodes N are labeled with the clauses, and A is the set of arcs with $(C, D) \in A$ if and only if $\mathcal{P} \Rightarrow_R D$ with $C \in \mathcal{P}$ and $R \in \mathcal{R}$. As such the distance of a clause node C to the closest input clause node corresponds to the inference depth of C. $\mathcal{R}^{SNE}$ explores this search space in breadth, deriving all clauses at a given depth i in the i-th loop iteration. $\mathcal{R}^{GCA}$ on the other hand relies much more on heuristics, allowing it to derive clauses at a depth d while there are still underived clauses at some depth $d' < d$. On typical reasoning problems this enables the given-clause algorithm to find a proof fast due to not having to bother with irrelevant inferences. On QA problems however there is a large amount of irrelevant input clauses and inferences compared to the small unsatisfiable subset required for a proof. This decreases the probability that the heuristic clause selection picks the correct clauses, which in turn increases the risk that the given-clause algorithm wastes time exploring parts of the search space that are deeper than the unsatisfiable subset.

If a problem requires splitting, for example, when the hyper tableaux calculus or the model evolution calculus have to deal with non-Horn clauses, then this adds the complication of having branches in the derivation. Treating multiple branches simultaneously causes implementational difficulties, so both $\mathcal{R}^{GCA}$ and $\mathcal{R}^{SNE}$ (as in the case of E-Darwin and E-KRHyper) will likely pick branches heuristically and consider them case by case. The presence of non-Horn clauses is therefore unlikely to give an advantage to one strategy over the other. However, as calculi show great variety in how they split, if at all, this remains speculative with respect to provers in general.

We expect the semi-naive evaluation to be superior on problems as they occur in LogAnswer, Horn problems that are large enough to distract heuristic clause selection methods, yet with proofs that are sufficiently shallow so that the prover does not have to explore the deeper search space levels where the amount of derived clauses becomes unmanageable.

9.3 Evaluation

Our evaluation is an updated version of the testing we did in [8], this time including E-Darwin, as the main purpose is to compare it to E-KRHyper. The other provers are included mainly to provide an overview of the performance of

ATP	solved	$\bar{t}$	TPTP
E 1.4	1,685 (93.4%)	1.64 s	49%
E-Darwin 1.4	1,511 (83.7%)	11.67 s	28%
E-KRHyper 1.3	1,728 (95.7%)	1.53 s	27%
E-KRHyper 1.3 (opt)	1,728 (95.7%)	1.15 s	27%
iProver v0.8.1	1,537 (85.2%)	7.19 s	38%
Metis 2.3	347 (19.2%)	6.48 s	26%
Otter 3.3f	1,728 (95.7%)	9.28 s	23%
SInE 0.4	1,587 (87.9%)	7.96 s	19%
Vampire 0.6	1,695 (93.9%)	3.45 s	53%

Table 9.1: Evaluation results for ATP systems on LogAnswer problem test set

ATP systems on LogAnswer problems in general, although this way we can also evaluate whether their results are consistent with our analysis of the differences between the given-clause algorithm and semi-naive evaluation. We chose all FOL theorem provers that had participated in both the CNF (clause normal form problems) and the FOF (first-order formula problems) divisions of the CASC 2011, indicating that they are suitable for FOL problems in general. Also, only unique stand-alone provers were chosen, not meta-provers or variants of already chosen provers. All provers were the most recent versions available from their official sources at the time of this writing. Of these, E-KRHyper is the only one based on semi-naive evaluation, while all the others use some form of the given-clause algorithm.

In addition to the above we also included a special case in the test line-up, the *SInE* [HV11] system, a preprocessor that heuristically selects a subset of a given input problem and then tests only this subset with the E prover. Its modus operandi is different from that of a typical ATP system, and it is included here mostly because it was part of our original test series [8]. Heuristic clause selection and SInE will be in the focus later on, see Section 12.2.2.

The systems were tested on our standard test computer with an Intel Q9950 CPU with 2.83 GHz. Each system used one CPU core to process one test problem. The time limit per problem was set at 60 seconds and the memory limit at 1GB of RAM. Some ATP systems adapt their strategies and heuristics based on resource limits, so where applicable, the systems were informed of these limits via parameters. The usage of these resources was monitored by the *TreeLimitedRun* program[6] that was developed for TPTP testing and the CASC. Otherwise all systems ran in their default configurations. An exception was a second test run we conducted for E-KRHyper, where the starting weight for the iterative deepening was increased to 12, a value we know to be more suitable for LogAnswer problems. Neither E-KRHyper run used any of the upcoming improvements specifically for the embedding as a reasoning server in LogAnswer, as their effect will be evaluated separately.

Table 9.1 summarizes the results. The leftmost column *ATP* gives the name and version number of the respective prover. The second column *solved* states the number of problems solved by that prover, also giving this as a percentage

[6] `http://www.cs.miami.edu/~tptp/CASC/J4/TreeLimitedRun.c`

of the total of 1,805 problems. The third column $\bar{t}$ states the average time the respective prover required for a proof. The final column TPTP is not a result of our testing, rather it states what percentage of the full TPTP the given system solves according to the official listings.[7] It serves to provide an idea of how the provers perform in general.

As can be seen, most provers solved a substantial number of the problems, with Otter and E-KRHyper tied at a maximal 95.7% of the set. The differences are more significant regarding the time. E-KRHyper required 1.53 seconds on average in its default configuration and 1.15 seconds with the optimized weight. Both settings were faster than any of the other systems, although the runner-up E comes close with 1.64 seconds. E-KRHyper clearly outperforms E-Darwin, both by solving more problems and by doing it faster, on average solving a problem in 13.1% and 9.9% of the time E-Darwin required. Given the similarities between the systems, it is clear that the search heuristics in E-Darwin have difficulties making the relevant choices in the face of the large number of clauses, while the relatively rigid breadth-first approach in E-KRHyper finds the proof earlier.

All tested given-clause systems are slower than the semi-naive E-KRHyper, which is consistent with our analysis. However, the average times vary greatly, and some of the given-clause systems come much closer to E-KRHyper than others. Obviously the given-clause algorithm can be adapted to such problems reasonably well. Indeed, with ideal heuristics the given-clause algorithm could potentially pick the shortest inference sequence leading to the proof, and thereby outperform the semi-naive approach which always evaluates the search space in breadth, most of which is irrelevant for the proof.

A curious observation is that SInE, which in this version employs E, showed a worse performance than E alone, both regarding the speed and the number of solutions. The SInE algorithm is generally a very promising approach to dealing with large problems. Section 12.2.2 investigates the difficulties apparently encountered on the LogAnswer problems.

While it is interesting that the semi-naive prover outperformed all given-clause provers on the test set, in particular given that on general TPTP testing the ranking of the systems is quite different, and while these results are consistent with our theoretical considerations, it would nevertheless be premature to attribute the performance differences solely to the choice of basic strategy. Prover performance depends on more factors than this, for example the individual calculus, redundancy handling, the programming language and the quality of the implementation. Only in the case of E-Darwin and E-KRHyper is there enough similarity between the systems to draw this conclusion with some certainty. Regarding the other provers more in-depth analysis would be required. Also, it should be noted that our evaluation used a set of fairly homogeneous problems. Naturally this choice was made as the test served to evaluate provers for LogAnswer, but this restriction limits the validity of conclusions for reasoning problems in general.

Nevertheless it was the speed differences, both as shown here and of course in the earlier testing which had similar results, which convinced us that E-

[7] The official TPTP numbers were obtained on 1 May 2012 from:
`http://www.cs.miami.edu/~tptp/TPTP/Results.html`
These numbers may have been produced using different system versions, but the performance usually does not change drastically between versions.

KRHyper was the correct choice for LogAnswer. These differences between the provers may appear trivial considering that most systems solved around 90% of the problems and that the official TPTP testing and CASC allow considerably more generous limits. However, the usage scenarios of QA systems are often restricted in their response times. A web-based QA system that competes with conventional search engines should ideally offer almost instantaneous replies, though we generally hold five seconds to be acceptable. In a QA competition like CLEF there is more time, usually around 10 minutes per question. While this may appear to be ample time, for a full logical processing of a question a prover has to carry out proof attempts for hundreds of answer candidates, and several per candidate when using robustness enhancements, see Section 13.1. This means up to 600 attempted proofs per question, in extreme cases more. It becomes clear that even small differences in the proof time quickly add up under these circumstances to such a degree that a slightly slower prover may become completely unsuitable. When minimal response times are required, as in the web-based scenario, then a full logical processing of all candidates is rarely possible. Here LogAnswer relies more on the ML-based ranking of candidates so that at least the most promising ones can be tested by the prover. In this situation small variances in the proof times of provers decide how many candidates can be tested with each prover, thus again the differences are not as insignificant as they may appear.

Regarding the comparison of the semi-naive evaluation in E-KRHyper and the given-clause algorithm in E-Darwin, the former appears to gain an advantage with its strategy on the QA tasks as encountered in LogAnswer, which is consistent with the theoretical considerations. The semi-naive strategy also preserves refutational completeness, an advantage over incomplete selection heuristics like SInE - which may nevertheless have a place in QA, as will be discussed in Section 12.2.2. At the same time the semi-naive evaluation faces difficulties regarding complex proofs due to its exponential generation of clauses, a problem that will likely prevent E-KRHyper from achieving the same general theorem proving capabilities as the best given-clause provers. As such the semi-naive evaluation can be seen as an aid when dealing with certain problem classes. In theory a given-clause algorithm prover could employ the semi-naive evaluation with little implementational effort. A simple approximation would consist of adjusting the clause selection heuristics in such a way that the set of support becomes a simple queue operating by the *First In, First Out* principle. A closer imitation would add a temporary set for the immediate storage of inference conclusions, before they reach the set of support: As soon as the set of support is empty, all stored inference conclusions are moved to the set of support. Likewise, a prover like E-KRHyper could use an approximation of the given-clause algorithm by not emptying its entire set of inference conclusions in every round, instead using selection heuristics to pick only one clause. However, in practice most ATP implementations are so intricate that even minor changes are rarely simple. For LogAnswer we therefore chose to integrate the ATP system that was better suited in the first place.

Chapter 10

Indexing and Subsumption of Multi-Literal Clauses

This chapter deals with the problem of indexing clauses with multiple literals. Most inferences only need to find single literals or subterms within such clauses, and thus they can use normal term-indexing by only considering the terms of interest. However, when (non-proper) subsumption between such clauses is to be computed, then the issues of both indexing and searching multiple literals that belong together cannot be avoided any more.

The first section describes the problems posed by multi-literal clauses. The second section then describes how indexing and subsumption of such clauses is handled in E-KRHyper. The final section is an evaluation of the indexing described here.

10.1 The Problem of Multi-Literal Clauses

As mentioned in Section 7.4.1, indexing techniques like discrimination trees index by terms and literals, and it is not trivial to extend this to clauses. Yet automated theorem provers have to index entire clauses in order to compute (non-proper) subsumption, which is an important countermeasure against excessive clause generation. In the case of unit clauses E-KRHyper uses the regular indexing as described before, with the predicate symbol treated as a function symbol for the purpose of forming a flattened term by which to determine the position in the discriminiation-tree, and equational atoms being indexed in both orientations. This method fails for clauses with multiple literals. Technically a flattened term could be computed for a clause by considering all of its symbols (including the negation $\neg$ and the disjunction $\vee$) as function symbols, then treating the entire clause as a "term" and flattening it. As there is no fixed ordering of literals due to the associativity and commutativity of the disjunction $\vee$, this results in a problem similar to the commutativity of equations in that different flattened terms are possible for a single clause. However, carrying over the solution of simply indexing all permutations is unfeasible, since in the worst case of a clause consisting of n non-orientable equational literals there would be $n! \cdot 2^n$ variations to index. Picking only one constellation for indexing and then performing multiple searches for all permutations instead is equally prohibitive.

Generally no ideal solution exists, since determining subsumption between clauses is an NP-complete decision problem [KN86]. In practice most clauses are not as unwieldy as the worst case, and by using additional filtering methods, and in the case of E-hyper tableaux further restricting the subsumption testing to non-proper subsumption, the problem can be rendered more benign. The earliest version of E-KRHyper was developed under time constraints and hence employed a relatively crude solution where simple matching of predicate symbol occurrences was used to determine a set of subsumption candidates, which then were compared to the search clause one by one, literal by literal [14]. This needed to be replaced by a better solution, both for a general performance improvement, but also in particular with regards to the large clause sets common in QA.

Modern ATP systems usually employ a method that is basically similar to the provisional two-phase approach of the E-KRHyper prototype, yet significantly more refined. First subsumption candidates are retrieved using a series of filtering criteria, for example a comparison of size and weight measures of clauses, symbol occurrences, and literal depths. Such criteria can be arranged and searched in tree-like indexing structures [Sch04]. Discrimination trees have also been used as a filtering criterion in the form of finding candidates which have at least one literal that is an instance or generalization (backward or forward subsumption) of the search clause [Tam98]. In the second phase the candidates are compared to the search clause one by one in full subsumption tests. This is often done with a variation of the Stillman-algorithm [Sti73] which extends a substitution while comparing the clauses literal by literal, and which backtracks in the case of a substitution clash.

The complexity of multi-literal clause subsumption has led to attempts to balance its cost against the savings due to omitted clauses. The DISCOUNT-loop algorithm mentioned in Section 9.2 was introduced as a refinement of the given-clause algorithm; its main improvement consists of excluding the set of support from subsumption operations. Backward subsumption in particular is sometimes seen as having dubious value [Tam98]. The Vampire ATP system used to employ a unique approach [RV03] where each literal of the subsuming clause was translated into an SQL-query to an index database, the results of which were then joined. However, in his tutorial *"First-order theorem proving and Vampire"* at CADE 2011, Andrei Voronkov revealed that the most recent version of Vampire no longer uses any backward subsumption, as experiments showed it was not worth the computational effort.

10.2 The Solution in E-KRHyper

For E-KRHyper we have devised the following method as a replacement for the provisional solution in the prototype. To the best of our knowledge our approach is unique, although implementation details like this are rarely published. The E-KRHyper method is closer to the aforementioned Vampire approach than to the conventional two-phase method, since we directly retrieve subsumed clauses rather than retrieving candidates which require another clause-to-clause subsumption test. However, unlike the Vampire approach we do not use an SQL-database with join operations, instead we perform sequential searches in perfect discrimination trees while extending substitutions. The development of our solution was motivated by several considerations.

Firstly, the clause sets for QA-related reasoning problems are abnormally large, and any way to reduce the number of clauses must be considered, which includes the subsumption between multi-literal clauses. Also, normal reasoning problems start with relatively small sets of clauses and then result in more and more clauses during the derivation. Under such circumstances it may be acceptable to focus only on forward subsumption, which prevents the addition of new clauses, while omitting backward subsumption, which would delete clauses already in the system. The hope then is that forward subsumption prevents the originally manageable set of clauses from growing too large. QA-related clause sets on the other hand are (too) large right from the start, and therefore it is also necessary to consider backward subsumption as a means of discarding existing clauses.

Secondly, the semi-naive evaluation strategy of E-KRHyper, while effective on problems with QA-characteristics, features no passive set of support that could be excluded from costly subsumption computations, in the manner of the DISCOUNT-loop. To adopt the terminology from the given-clause algorithm, all clauses in E-KRHyper are *usable*, so the indexing and subsumption of multi-literal clauses must remain effective while covering all clauses in the system.

Finally, unlike most theorem provers E-KRHyper attempts to decide reasoning problems by producing a model for satisfiable input. For this to succeed the prover must terminate on satisfiable input by finding a derivational fixed-point. Unlike a prover which only cares about finding contradictions, E-KRHyper must therefore also aggressively prune the search space by removing as many clauses as possible. While this is less of an issue with QA-related problems where we are only interested in refutations, it provides a motivation for an indexing and subsumption method that is effective for theorem proving in general.

The idea behind our solution is to integrate the computation of substitutions into the search for subsumption candidates, both in order to avoid the subsequent individual clause-to-clause subsumption tests, and to avoid completely recomputing similar substitutions for different clauses by sharing parts of the computation. The method mainly supports non-proper subsumption, because this is what the E-hyper tableaux calculus prescribes. With minor adjustments the same algorithm is used both for forward and backward non-proper subsumption. It can also be adapted to proper backward subsumption reasonably well, but not to proper forward subsumption. To determine whether a given clause C can be used for forward or backward subsumption with respect to indexed clauses, instead of first filtering and then testing candidates, our approach uses perfect discrimination trees to search by all literals of C while integrating the filtering and testing into this search.

Every clause is indexed once by each of its non-equational literals and twice by each of its equational literals, covering both orientations as described above. Also, every clause is equipped with a boolean *search flag*, which is used in several steps of the algorithm. As default the search flags are turned off (set to *false)*. Each index node keeps a list of pointers to the search flags of the clauses stored in the subtree below it. Perfect discrimination trees were chosen so that the substitutions can be computed and checked during the search in the trees. That way a search can be stopped as soon as a conflict arises while trying to extend a substitution. Also, all clauses found at one particular leaf during a search are immediately known to have been found by one specific substitution. This greatly simplifies further searches, as will be seen below.

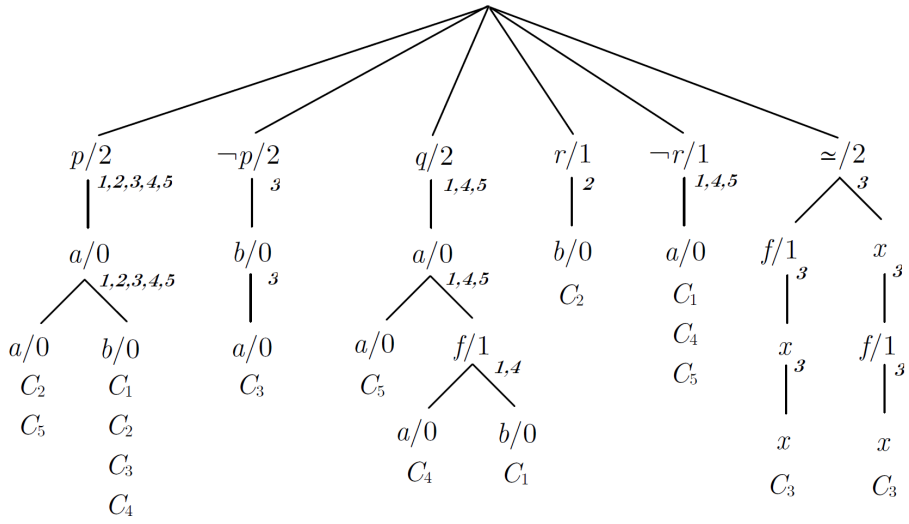


Figure 10.1: Perfect discrimination tree subsumption index for multi-literal clauses: an italic number i at a node indicates that the node carries the search flag of clause C_i.

As an example, consider the index in Figure 10.1 for the following set of clauses:

C_1: $\{p(a,b), q(a,f(b)), \neg r(a)\}$

C_2: $\{p(a,b), p(a,a), r(b)\}$

C_3: $\{f(x) \simeq x, p(a,b), \neg p(b,a)\}$

C_4: $\{p(a,b), q(a,f(a)), \neg r(a)\}$

C_5: $\{p(a,a), q(a,a), \neg r(a)\}$

The basic algorithm for non-proper back subsumption in E-KRHyper is shown in the pseudo-code of Algorithm 10.1; however, some explanatory words are likely necessary.

Given is a clause C consisting of literals $L_1, \ldots, L_n$, and the algorithm searches for all indexed clauses non-properly subsumed by C. The function SEARCH recursively iterates over the non-properly subsumed clauses and collects them. In every iteration it takes one literal $L_i \in C$, a substitution σ and a set of clauses *previousClauses*, and then it searches the index for clauses that contain an instance of $L_i\sigma$. In the initial call to SEARCH, which searches for the first literal L_1, the substitution σ and the set *previousClauses* are empty, and all clauses found for L_1 will be accepted as preliminary candidates. In later calls σ is the substitution under which the *previousClauses* have been found to contain $L_1\sigma, \ldots, L_{i-1}\sigma$ in previous iterations for the previous literals of C.

Initially the SEARCH function accesses the index to retrieve clauses containing an instance of L_1, saving them in the set of *candidates*. Before and after this actual index search, the search flags of the *previousClauses* are turned on and

Algorithm 10.1 E-KRHyper uses multiple searches in perfect discrimination trees for backward non-proper subsumption.

```
C := the query clause;
// C = {L_1, ..., L_n}
// The task is to find all indexed clauses non-properly subsumed by C.

function SEARCH(i, σ, previousClauses)
    // previousClauses are the previously found clauses
    // containing L_1σ, ..., L_{i-1}σ.
    // The task is to find every indexed clause D
    // containing an instance of L_iσ,
    // with D ∈ previousClauses if i > 1,
    // and then recursively continue with L_{i+1}.

    turn on search flags of previousClauses;
    candidates := search the index by L_iσ;
    turn off search flags of previousClauses;
    // candidates = {result_1, ..., result_m}
    // with every result_j = (δ_j, clauses_j) so that
    // 1. L_iσδ_j = L for some literal L shared by all clauses_j, and
    // 2. clauses_j ⊆ previousClauses if i > 1

    if i = 1 then
        candidates := filter(candidates)
    end if
    if i = n then
        return candidates
    else
        subsumed := {};
        for all result_j = (δ_j, clauses_j) do
            subsumed_j := SEARCH(i + 1, σδ_j, clauses_j);
            subsumed := subsumed ∪ subsumed_j;
        end for
        return subsumed
    end if
end function

subsumed := SEARCH(1, empty substitution, {})
```

then off again. While in this search for the first literal the *previousClauses* are empty and the index search ignores these search flags entirely, the search flag optimization ensures that in subsequent calls the index search will only access index subtrees which have at least one activated search flag, indicating that they contain clauses which have already been found for the previous literals. Along the same lines, when reaching a leaf only those clauses with an activated search flag will be gathered as candidates. As there may be different instantiating substitutions if L_1 is not ground, the resulting set of *candidates* consists of several *result*-groups. Each of these is comprised of a substitution and those clauses that have a literal instance of L_1 when using that particular substitution.

For example, let $C = \{p(x,y), q(z,f(x)), \neg r(a)\}$, then an initial search in the index of Figure 10.1 by $p(x,y)$ returns:

candidates (for $L_1 = p(x,y)$) =

***result*$_1$:** $\sigma_1 = \{x \leftarrow a, y \leftarrow b\}$
$D_{1,1} = C_1 = \{p(a,b), q(a,f(b)), \neg r(a)\}$
$D_{1,2} = C_2 = \{p(a,b), p(a,a), r(b)\}$
$D_{1,3} = C_3 = \{f(x) \simeq x, p(a,b), \neg p(b,a)\}$
$D_{1,4} = C_4 = \{p(a,b), q(a,f(a)), \neg r(a)\}$

***result*$_2$:** $\sigma_2 = \{x \leftarrow a, y \leftarrow a\}$
$D_{2,1} = C_2 = \{p(a,b), p(a,a), r(b)\}$
$D_{2,2} = C_5 = \{p(a,a), q(a,a), \neg r(a)\}$

Note that the clause C_2 is found twice, as $D_{1,2}$ and as $D_{2,1}$, because σ_1 and σ_2 cause L_1 to instantiate to two different literals of C_2.

In this initial iteration for the first literal a quick filtering phase now prunes the results using various criteria, most of which have already been precomputed to some degree during the creation of each clause:

Clause depth: A clause cannot non-properly subsume a clause of lesser clause depth.

Literal signs: A clause with positive literals cannot non-properly subsume a clause without positive literals, and a clause with negative literals cannot non-properly subsume a clause without negative literals.

Symbol count: A clause cannot non-properly subsume a clause with fewer unique symbols.

Predicate symbols: A clause cannot non-properly subsume a clause with different predicate symbols.

There is also an optional criterion of clause length. In its strictest form it requires the subsumer and the subsumee to have the same number of literals. It can be weakened or omitted depending on whether clauses are seen as sets or as multisets, and also to allow (proper) subsumption. This works in conjunction with a final filtering at the end of the algorithm, and the matter will be discussed further at the end of this section.

In the example, $D_{2,2}$ is discarded due to its insufficient clause depth. $D_{1,2} = D_{2,1}$ has only positive literals and both its occurrences are discarded due to

the criterion of literal signs. $D_{1,3}$ is removed as it fails both the symbol count criterion and the predicate symbols. Any result substitution σ_i left without clauses is discarded as well, which leaves these pruned results:

candidates (for $L_1 = p(x, y)$) =

***result*$_1$:** $\sigma_1 = \{x \leftarrow a, y \leftarrow b\}$
$D_{1,1} = C_1 = \{p(a, b), q(a, f(b)), \neg r(a)\}$
$D_{1,4} = C_4 = \{p(a, b), q(a, f(a)), \neg r(a)\}$

The SEARCH function then recursively invokes itself on the pruned results, which effectively means that it descends depth-first in the tree of extending substitutions. For each *result*-group $result_j$ the search flags of the found clauses are activated, and then SEARCH is called for the next literal and the extended substitution $\sigma\delta_j$, trying to compute that subset of the $clauses_j$ in $result_j$ which also instantiate the next literal of C under a common substitution that is an extension of $\sigma\delta_j$.

In the example SEARCH is thus called for the second literal of C with the substitution $\sigma_1 = \{x \leftarrow a, y \leftarrow b\}$, i.e. $q(z, f(a))$, and the $clauses_1 = \{C_1, C_4\}$. The search flags of these clauses are activated. Only one clause is retrieved, C_4:

candidates (for $L_2\sigma_1 = q(z, f(a))$) =

***result*$_1$:** $\sigma_1\delta_1 = \{x \leftarrow a, y \leftarrow b, z \leftarrow a\}$
$D_{1,1} = C_4 = \{p(a, b), q(a, f(a)), \neg r(a)\}$

The clause C_1 fails to instantiate the search literal and thus no longer appears. The search flags are turned off again and the function SEARCH is now called a last time for the third literal of C and the extended substitution $\sigma_1\delta_1$, hence with the search literal $\neg r(a)$, and $clauses_1 = \{C_4\}$. Only the search flag of C_4 is activated, and therefore only C_4 is retrieved. While both C_1 and C_5 would also have instantiated the search literal, their search flags are not active, and thus they are not valid candidates, having failed to instantiate some earlier search literal.

With the last literal of C instantiated, C_4 is now known to be an instance of C, so C_4 and the instantiating substitution are returned in *subsumed*. In E-KRHyper these clauses are immediately marked as redundant, although technically the algorithm also allows a mere retrieval of instances for other purposes. Further calls to SEARCH may now add more non-properly subsumed clauses by testing other substitutions for earlier literals in previous iterations, although in our example this is not necessary.

The algorithm for non-proper forward subsumption is analogous, except that generalizations are searched instead of instances. The substitutions that are extended over the calls to SEARCH are the ones that may instantiate the indexed generalization candidates to become equal to C. This means that the search literals remain in their original form as taken from C, and instead the index search within the perfect discrimination trees uses the growing substitutions to instantiate variable nodes on the fly. That way substitution clashes can be detected during the search for a particular combination of literal and substitution in the respective branch.

10.3 Modifications

When a clause has been found for all literals of C with one common substitution, then it may be subject to further tests, depending on the specific notions of clauses and subsumption in use. Optional filters can adjust the algorithm to regard the clauses as sets or as multisets. When considered as sets, a clause like $\{p(a,x), p(y,b)\}$ can non-properly subsume $\{p(a,b)\}$, a behaviour that may be undesirable, given that the subsuming clause is arguably more complex. On the other hand, $\{p(x,y)\}$ may non-properly subsume $\{p(a,x), p(y,b)\}$ when seen as sets, which may be advantageous. The set treatment necessitates loosening the clause length criterion (see above), because a requirement of strict length equality allows neither of the special subsumption examples. Overall our experimenting with the TPTP shows no significant advantage to any of these settings.

On a related note it should be remarked that the algorithm as given may in some cases result in proper set subsumption, even when strict length equality is required. In the backwards direction this is the case when multiple literals of the search clause C subsume a single literal of an indexed clause D. For example, let $C = \{p(a,x), p(y,b)\}$ and $D = \{p(a,b), p(c,d)\}$. The clauses fulfill all filtering criteria, and after two iterations both literals of C will have been found to subsume the first literal of D, leaving the second D-literal $p(c,d)$ unsubsumed. Usually this poses no problem when the goal is to discard as many clauses as possible, since D as a whole is subsumed by C. When strict non-proper subsumption is desired, though, then a final check is necessary in which the instantiating substitution is applied to C and both clauses are tested for equality. The substitution ensures that if C truly non-properly subsumes D, then $C\sigma$ is equal to D modulo the order of literals and the orientation of equations. This makes it trivial to sort the clause literals of both clauses using a special term ordering that treats variable names like constant names, and which is thus total even on non-ground terms.[1] If $C\sigma$ non-properly subsumes D, then the sorted clauses $C\sigma'$ and D' are equal even regarding the literal ordering and the equation orientations, which allows a linear comparison of the clauses literal by literal. In the example we obtain $C\sigma = \{p(a,b), p(a,b)\}$, which is quickly determined to differ from D. On the other hand, if $C = \{p(x,y), f(x) \simeq y\}$ and $D = \{z \simeq f(u), p(u,a)\}$, then by substituting and sorting we obtain $C\sigma' = \{p(u,a), f(u) \simeq z\}$ and $D' = \{p(u,a), f(u) \simeq z\}$, whose equality is easily determined.

The algorithm for non-proper backward subsumption can be adapted reasonably well to proper backward subsumption, i.e. given a clause C, find all clauses D with $C\sigma \subset D$. In that case we are essentially searching for any clause D which has a subset that is non-properly subsumed by C. The algorithm can achieve this by loosening the filtering criteria of clause length and predicate symbols, and by adjusting the set/multiset-setting, thereby allowing a clause C to subsume a clause D once every literal in C has found an instance in D, regardless of whether there are still other unsubsumed literals in D. The same

[1] Naturally such an ordering is of limited use in most cases, such as comparisons between arbitrary clauses, as variables can be renamed and hence their treatment as constant names depends on the current naming. However, for any given variable naming of a clause this modified ordering will result in an unambiguous canonical ordering of literals and orientation of equations. Thus, when two clauses have the same variable names due to a unifying substitution, as is the case here, then if they are variants after the substitution they can be ordered into the identical form, making their comparison trivial.

test type	problems solved
default settings	5,986 (38.5%/30.8%)
no multi-literal subsumption	4,578 (29.4%/23.5%)
no multi-literal back subsumption	5,770 (37.1%/29.7%)
proper back subsumption	5,976 (38.4%/30.7%)

Table 10.1: Test results for E-KRHyper with various subsumption settings on the TPTP; percentages refer to the FOL test set and to the total TPTP respectively.

does not hold true for proper forward subsumption, because in that case the algorithm cannot simply attempt to match the literals of C one by one, since the intention is to find a subset of C that is non-properly subsumed by some clause. This could be done by forming the subsets of C and searching for a non-proper subsumer for each, but this is impractical for larger clauses. The reduction phases in E-KRHyper (see Section 7.3.2) already include unit clauses properly subsuming both unit and multi-literal clauses in the forward as well as in the backward direction, and in combination with the non-proper subsumption between multi-literal clauses these relatively efficient operations cover so many cases of subsumption that a complex search for the remaining cases is not worth the computational effort.

When using proper backward-subsumption, the optional clause length criterion after the initial search by the first literal L_1 is set to allow subsumees with more literals than the subsumer, whereas the default setting in E-KRHyper is to use non-proper subsumption and to require subsumer and subsumee to have exactly the same number of literals. In practice the stricter setting is more effective, as it prunes the candidate set to a greater degree, whereas the additional candidates allowed by proper subsumption make the search more costly, yet they rarely actually pass the full test to be subsumed.

10.4 Evaluation

We evaluated E-KRHyper on the TPTP v5.3.0 with various subsumption settings. The results are summarized in Table 10.1. Our standard TPTP test setup (see Section 7.5) was used. For a comparison the first test is a repetition of the initial results from Chapter 7, which had E-KRHyper running under default settings, with non-proper subsumption enabled for clauses of any length. 5,986 problems (38.5%) of the test set were solved, or 30.8% of the total TPTP. For the next test the multi-literal indexing and subsumption described in this chapter was disabled. Under these conditions E-KRHyper solved 4,578 of the 15,550 FOL-problems of the TPTP v5.3.0, which is 29.4% of the test set or 23.5% of the total TPTP with 19,446 problems. As even the restricted E-KRHyper still exploited subsumption and simplification when it involved unit clauses, the performance difference can be attributed solely to the lack of indexing and subsumption between clauses with at least two literals.

In the next test we evaluated the impact of forward versus backward multi-literal subsumption. The settings are very similar to the previous test, except that (non-proper) forward multi-literal subsumption remained enabled. 5,770

problems were solved, corresponding to 37.1% of the test set and 29.7% of the total TPTP. The difference to the default settings which used non-proper multi-literal subsumption in both directions is minor, forgoing non-proper multi-literal back subsumption resulted in only 216 problems fewer solved. The forward direction is thus responsible for the majority of the perfomance gain exhibited by the comparison between the first two test results. Nevertheless even the backward direction has a positive effect, albeit small, so we cannot confirm the negative views on back subsumption mentioned in Section 10.1.

The fourth test had E-KRHyper using proper back subsumption, while forward subsumption remained at the default non-proper setting. The result shows a minor decrease in solutions compared to the full default settings, which solved 10 additional problems, or 30.8% versus 30.7% of the total TPTP. This difference is too small to conclude a definitive advantage for either setting to backward subsumption. Differences in this range can occur even between repeated runs under the same settings, as minor fluctuations in the CPU load can slow down the derivation seemingly at random, resulting in problems with solution times close to the time limit sometimes being solved and sometimes not. Considering that the default settings allow both non-proper subsumption between clauses of any length and also units properly subsuming clauses of any length, it seems safe to say that this default covers such a large amount of subsumptions that the additional processing for proper subsumption between multi-literal clauses does not offer any noteworthy performance gains. The forward direction constitutes an analogous combination of non-proper subsumption between clauses of any length and proper subsumption of clauses of any length by units. As mentioned, the computation of proper multi-literal forward subsumption is significantly more complex than in the backward direction, so we feel justified in staying with the default here as well.

Chapter 11

Technical Aspects of Handling Large Problems

The information retrieval phase in LogAnswer (see Section 8.4) ensures that the individual reasoning problems presented to E-KRHyper consist of thousands of clauses, rather than the millions that make up the full knowledge base. While this places the problems in a range in which ATP systems can operate, their size is still orders of magnitude above the problems provers are intended for. With approximately 11,000 clauses a LogAnswer problem is around 200 times larger than the common TPTP problem with a median of 52 clauses (see Section 5.2.2). Furthermore, in many QA use cases the prover has little time to find a proof. Hundreds of reasoning problems must be processed within seconds to achieve a response time that is acceptable for a casual user, who is used to nearly instant results from conventional search engines.

A prover like E-KRHyper must therefore be adapted to handle the size of QA-related problems. These modifications of the prover can be divided into two groups. One side consists of technical optimizations of the implementation which harden the prover against the difficulties caused by oversized problems and which streamline the operation to ensure an efficient handling of such problems. Modifications of this first type are directly applied to the implementation of a specific prover. The other group are changes on the logical level, predominantly consisting of methods which attempt to reduce the clause sets even further, for example through a heuristic question-based clause selection. Modifications of the second type can be independent of a specific prover - for example, they can be implemented in a preprocessor - although for the sake of efficiency it often makes sense to integrate them into the ATP system.

This chapter will deal with modifications of the first type, i.e. the technical level, while the next chapter is dedicated to the logic level optimizations. Each of the following sections will detail one aspect of these adaptations as implemented in E-KRHyper.

11.1 Stability

As mentioned in Section 5.2.2, large reasoning problems can take a theorem prover to its limits and break its operation. Therefore one of the first measures

to take when adapting a theorem prover to QA is to ensure that the prover remains stable when handling such problems, irrespective of whether it can actually solve them.

In the case of an OCaml-based prover like E-KRHyper this means that special attention must be paid to all list-based operations. The functions in the `List`-module of OCaml are not tail-recursive. This is to make them more efficient on short and medium-sized lists, at the cost of using stack space proportional to the length of their list argument. On lists with several thousand elements the resulting stack space usage can exceed the system limits, causing the prover to crash without any error message. The cut-off point for safe usage is difficult to predict, as there is no exact documentation on this issue. Therefore the prover should be stress-tested with very large reasoning problems, for example the Cyc-related TPTP problems *CSR025+1* to *CSR074+6*. While no ATP can actually reason effectively on the largest of these problems without applying some form of clause selection, simply attempting to load such a problem and to start a derivation can already show whether the prover remains stable or is overwhelmed by the sheer amount of clauses. Wherever `List`-based functions fail, they should be replaced by bespoke tail-recursive functions.

Other limitations can be found this way as well, like data structures with a fixed limit on the number of elements they can store, for example symbol tables with a maximum size, or arrays for the computation of substitutions which assume at most n variables.

Another point to consider is that as the reasoning problems grow, the computational burden does not increase in a similar degree for all parts of the prover. Rather, some data structures may see an explosive growth in the number of elements to store while others are barely affected. For example, when developing the ontology underlying a knowledge base it can make sense to keep the number of relations low, resulting in a small number of predicates. This simplifies the formulation of reasoning rules. The current FOL knowledge base of LogAnswer uses a fixed number of only 86 different predicate symbols for its 11,000 atoms, about 128 atoms per predicate. Compare this to the predicate inventory of Cyc that has grown in an ad-hoc fashion: In its largest TPTP version Cyc has 204,678 predicates for 5,328,208 atoms, 26 atoms per predicate. This means that depending on the ontology, the indexing trees may scale more or less well as the reasoning problems grow. As Cyc-based reasoning problems grow, for example by taking subsets of increasing size, the number of indexing trees (one per predicate) will grow at a similar rate, while the storage requirements for the individual indexing trees do not change much. The smallest TPTP subset of Cyc has 269 predicates for 1,964 atoms, 7.3 atoms per predicate. Thus, while the full TPTP version of Cyc above is approximately 2,700 times larger than the smallest subset, the number of atoms per predicate has not even quadrupled. The implementation of an individual indexing tree may therefore not require any changes to cope with larger Cyc problems. Instead it is the number of trees that increases. With LogAnswer problems on the other hand all atoms, no matter how large the problem, have to be stored in the indexing trees of the same 86 predicates. Increasing the problem size by n atoms will therefore increase the storage requirement of the average tree by $\frac{n}{86}$, with a corresponding increase in branching. In practice the problem is even more severe as some predicates are significantly more common than others, causing a select few indexing trees to store the bulk of the atoms. If child nodes are implemented in a manner that

does not scale well to these increases, for example by being stored in lists as in the early E-KRHyper instead of in hash tables, then indexing operations can become a bottleneck.

11.2 Reusing Input

The normal modus operandi of an automated theorem prover is to be started anew for each reasoning problem, and then the prover terminates once it has arrived at some result or reached some resource limit. The prover is therefore oblivious of any problems encountered earlier, and if problems share some set of axioms or a knowledge base, then these shared parts have to pass through any preprocessing and indexing repeatedly.

This is inefficient, and particularly so in LogAnswer, where the background knowledge is shared between all reasoning problems, forming about 98% of each problem. Starting E-KRHyper and loading only the background knowledge base takes 0.6 seconds on our test system. Recall that the prover on average requires 1.15 seconds to solve one of our LogAnswer test problems (see Section 9.3), and it becomes clear that here is an opportunity to reduce the processing time significantly. By keeping E-KRHyper operational between problems, the background knowledge has to be loaded and indexed only once. When this is done, loading a problem can be reduced to loading the FOL representations of the query and the answer candidate. After the proof attempt the problem-specific clauses must be retracted again to clear the prover for the next problem.

To support this operation the layered indexing (see Section 7.4.1) of E-KRHyper has been converted into allowing the saving and loading of clause set states. Recall that the index layering was originally intended to speed up backtracking in tableaux with multiple branches: Internally every index may consist of multiple trees, the layers, and at every choice point a layer is added. Index searching accesses all layers, while index insertion only writes into the most recent tree, the top layer. During backtracking the invalid layers are simply dropped by dereferencing. This layering structure has been utilized for state handling in the following way: Let $\mathcal{I}$ be a layered index with layers $L_1, \ldots, L_n$, where L_n is the most recent layer. An *index state* of I is a set of layers $S = \{L_1, \ldots, L_k\}$ with $k \leq n$. E-KRHyper implements an *index state stack* $\mathcal{S}$ of index states $S_1, \ldots, S_m$ and two operations:

***save state*:** Given $\mathcal{I} = L_1, \ldots, L_n$ and $\mathcal{S} = S_1, \ldots, S_m$, save the current index state by pushing $S_{m+1} = \{L_1, \ldots, L_n\}$ onto the index state stack $\mathcal{S}$ so that $\mathcal{S} = S_1, \ldots, S_m, S_{m+1}$, and add a layer L_{n+1} to $\mathcal{I}$ so that $\mathcal{I} = L_1, \ldots, L_n, L_{n+1}$.

***load previous state*:** Given $\mathcal{I} = L_1, \ldots, L_n$ and $\mathcal{S} = S_1, \ldots, S_m$, pop $S_m = \{L_1, \ldots, L_k\}$ from $\mathcal{S}$ so that $\mathcal{S} = S_1, \ldots, S_{m-1}$, and remove $L_{k+1}, \ldots, L_n$ from $\mathcal{I}$ so that $\mathcal{I} = L_1, \ldots, L_k$.

The states and the index use references to share the same layer structures, so there is no need to store copies of index trees. The stack-based state system is fairly simplistic in that a saved state can only be reloaded once. Also, if we wish to load a saved state S_i that is not the most recently saved state S_m, then the states $S_m, S_{m-1}, \ldots, S_{i+1}$ must first be loaded in this backwards chronological

order until S_i is the most recent state on $\mathcal{S}$, and the states $S_m, S_{m-1}, \ldots, S_{i+1}$ are lost after this. However, this is sufficient for our purposes and the implementation is efficient.

The operation of E-KRHyper when having to solve a series of LogAnswer-related reasoning problems $P_1, P_2, \ldots$ thus has the following steps:

1. start E-KRHyper and load the background knowledge base,

2. *save state*,

3. load and process P_1,

4. *load previous state*,

5. *save state*,

6. load and process P_2,

7. *load previous state*,

8. *save state*,

...

Once a previous state has been loaded, it must be saved again (steps 5 and 8) if it is necessary to return to that state again later on. This is because the *save state* operation adds a layer to the index, and if we skipped step 5, then the clauses of P_2 would be written into the same layers as the background knowledge, making it impossible to retract these problem-specific clauses after the processing of P_2. As the retracting is done by dereferencing, it requires only negligible time.

11.3 Continuous Operation

The previous section already introduced the notion of keeping a prover operational while processing multiple reasoning problems sequentially. Disregarding the aspect of reusing data, adapting a prover to continuous operation comes with its own set of requirements. The long-term stability must be ensured. A prover that was originally intended to terminate after one problem is likely to contain many single-use data structures. This is the case with configuration settings which normally do not change throughout the runtime once determined at start-up. Other structures may only be able to grow, for example symbol tables and term databases which are built as the input problem is parsed. Such structures are not necessarily designed in a way that they can be easily cleared of data that is no longer required for the next reasoning problem. Yet this is necessary: For example, when domain clauses are created for the enumeration of the Herbrand universe, the constants and function symbols are obtained from the symbol table, so any such symbols left-over from an earlier problem can have unpredictable consequences on the derivation for the current problem. The code of the prover must be thoroughly inspected and tested to make sure that the system can be reset into a clean state once one problem has been processed, and this resetting must be sufficiently efficient not to offset the time savings gained from not restarting the prover.

In the same vein the theorem prover must be examined for memory leaks. A prover can quickly consume large quantities of memory, which is why most modern provers allow the user to specify a memory limit so that the prover terminates as soon as it exceeds the limit, thus protecting the overarching computer system. A single-use prover can rely on the operating system to clear its memory after termination, but a continuously operating prover has to keep a tight control on its memory usage and explicitly free up unused resources. When a single-use prover is adapted to continuous use, efficient decisions made during the original design phase may result in memory leaks once the prover is embedded. The early version of E-KRHyper in LogAnswer was plagued by such memory leaks which required the prover to be restarted after every two hours of intense usage, like during the CLEF competition, in order to prevent it from having to use slow disk-based virtual memory via swapping. Thorough memory profiling was used to uncover problematic areas in the processing, and subsequently the code for the tableaux derivation was more cleanly encapsulated from the higher level routines that handle the processing of input, output and commands. The result is a prover that can fully reset its memory usage and thus remain operational indefinitely.

User-specified limits for memory and processing times are usually implemented in the form of a terminating interruption of the prover. During continuous operation this behaviour is likely undesirable, because a prover should not halt the processing of an entire series of problems just because one of them cannot be handled within the limits. E-KRHyper allows the user to specify in what manner to handle limit transgressions: The prover can terminate completely in the conventional way, or it can halt the current derivation while keeping the intermediate results computed so far, or it can stop the current derivation completely and discard any results. There is also an overriding interrupt method that will always terminate the prover, as this is required by CASC. This enables E-KRHyper to handle time and memory limits internally, for example by moving on to the next problem, while an overarching system like LogAnswer or the CASC framework can still halt the prover.

The capability for bidirectional communication with another system or the user becomes important when a prover is not merely supposed to process a predetermined series of problems and hand over the results, but when problems are selected outside of the prover during its runtime, for example by the user of a QA system, or when the results for one problem have some bearing on how to process the next one. This may be the case if some time limit of n seconds is considered acceptable for each of a series of m problems and one particular problem is solved in considerably less time. Then the limit for the next problem could be increased beyond n, or a problem that could not be solved within n seconds could be restarted if at the end of the problem series the overall limit of $m \cdot n$ has not yet been reached. A continuously operating prover should therefore offer the means to communicate over standard input and output streams, to accept commands at runtime and also to stand by in phases when it does not have to process any commands. Fortunately the original KRHyper was already equipped with bidirectional communication, and in E-KRHyper we extended this by adding support for different syntaxes, as the original's exclusive reliance on a prefix PROTEIN notation (similar to canonical Prolog) was very cumbersome, in particular for novice users.

11.4 Parallel Tableaux and Serialization

As a side effect of the encapsulation required for memory stability it became simple to allow E-KRHyper to maintain multiple clause sets and tableaux derivations in parallel. With the `change_clause_set` command the prover can create different proof environments with distinct clause sets and derivations, and it can switch between these. This makes it possible to keep different task-specific knowledge bases in the memory, so that the prover can adapt to queries. For example, this feature would be helpful in an adaptation of LogAnswer to multilingual usage: The prover could store background knowledge bases for different natural languages and then change between them as required by the incoming questions. It is also an important step towards parallel processing, although at the moment the OCaml support for this is rudimentary. The parallel tableaux feature is therefore still experimental and not used in practice by LogAnswer, although it may become important in the future.

Another related capability introduced in E-KRHyper is the serialization of indexed clause sets. This means that the prover can save a fully indexed knowledge base into a binary file. At any later time this file can be loaded to restore the knowledge base together with its index, without having to perform the lengthy indexing again. This can drastically cut the time for reloading. For example, the TPTP axiom set *CSR002+3.ax* (a subset of Cyc) takes 15 seconds to load and index conventionally, whereas its binary version can be loaded in circa one second. Ontologies like Cyc which include many multi-literal clauses result in large binary files when serialized: The size roughly quadruples compared to the original text-based specification in the TPTP, in this particular example expanding from 6 MB to 25 MB. The size increase is less severe for simpler ontologies. For example, the TPTP axiom set *MED002+0.ax*, a FOL version of the *MeSH* ontology,[1] consists of over 500,000 unit clauses. Loading and indexing this 49 MB file takes 45 seconds, while the serialized version weighs in at 109 MB and takes less than two seconds to load.

The savings may appear considerable, but in practice this feature has yet to find a good application in LogAnswer. Serialization optimizes the reloading of knowledge bases, whereas input reuse under continuous operation avoids reloading altogether and thus offers superior time savings. However, serialization has a potential usage as a memory saving alternative to parallel tableaux. Rather than keeping multiple knowledge bases in the memory, all knowledge bases may be stored in binary files on non-volatile media and then loaded and discarded as required, disburdening the RAM when they are not needed for the question at hand. As the memory usage of fully indexed complex ontologies tends to exceed the size of their original specifications by a factor of ten or more, the value of such considerations cannot be denied.

11.5 Evaluation

Much of what is described in this chapter relates to the stability of the prover during QA operation, and as such it is not easily quantifiable in an evaluation. However, the effects of reusing a knowledge base can be tested. For this purpose we again evaluated E-KRHyper on the test set of LogAnswer problems used in

[1] Medical Subject Headings: `http://www.nlm.nih.gov/mesh`

Section 9.3, but this time the prover was not restarted for each problem. Instead the background knowledge base was loaded into E-KRHyper once, and then the prover remained in continuous operation while processing all 1,805 problems. As usual the evaluation was performed on one of our standard test computers with an Intel Q9950 CPU with 2.83 GHz. A time limit of 60 seconds was used for each problem. This was handled internally by E-KRHyper, which aborted the processing of a problem as soon as the limit was reached, and which then restarted the clock and moved on to the next problem. The prover used the weight setting optimized for LogAnswer problems, see Section 9.3. Under these conditions E-KRHyper again solved 1,728 problems, 95.7% of the 1,805 problems. On average the prover required 0.99 seconds to solve a problem. Compare this to the evaluation with full restarting in Section 9.3, where E-KRHyper required 1.15 seconds on average. This corresponds to a speed-up by 13.9%. While this may appear disappointing considering that loading the knowledge base takes approximately 0.6 seconds, one has to keep in mind that there is still some overhead due to resetting E-KRHyper to the previous state between problems. Also, the average processing time is affected by outlier problems that require more than 20 seconds. If we consider the median solving time instead, then continuous operation with knowledge base reuse reduces the time from 0.7 seconds to 0.4 seconds, a speed-up by over 40%.

Chapter 12

Logical Aspects of Handling Large Problems

In the previous chapter we discussed methods to optimize the implementation of an automated theorem prover with respect to handling large reasoning problems. Now we will consider logic-based techniques with the same goal of improving prover performance on large input. They are applied to the logical input, and as such they are not strictly tied to a specific prover implementation, instead they can be placed in a separate preprocessor.

12.1 Redundancy Reduction

In the light of earlier chapters this is likely trivial, but one first step to consider is to reduce the input as much as possible before starting the derivation or even before handing it to the prover. This involves the exhaustive application of simplification and (non-proper) subsumption operations on the input, see Section 7.3.2 for a list of such operations. The goal is to remove tautologies and redundant or subsumed clauses from the input before they can participate in the reasoning. This is particularly useful when a large knowledge base is reused for many problems, as normally prohibitively expensive simplifications have to be applied to the knowledge base only once, and then they improve all future proof attempts with that reduced knowledge base.

However, in practice such redundancy elimination is unfeasible for the largest knowledge bases. Consider the Cyc ontology, which in its largest TPTP version (*CSR002+5.ax*) contains approximately 3.5 million clauses when transformed into CNF. Just loading this set into E-KRHyper will require about 20 minutes and close to 8 GB of memory, and that is with the special indexes for rewriting and multi-literal clause subsumption disabled, meaning that most reduction operations are not even available. Applying such operations on this scale would necessitate excessive hardware resources, rendering the proposition impracticable for the time being.

12.2 Axiom Selection

A different approach to reduce a large reasoning problem is to select a subset of its axioms or clauses in the hope that by using only this subset the reasoning can achieve the desired result in less time. Recall that a reasoning problem $P = (Ax, Con)$ consists of an axiom set Ax and a conjecture set Con (see Definition 3.2). The method used to perform a selection on P is called a *selection function*:

Definition 12.1 (Selection Function)**.** *A* selection function *sel is a mapping between axiom sets such that* $sel(Ax) \subseteq Ax$ *for any given problem* $P = (Ax, Con)$*. We extend this function to reasoning problems such that* $sel(P) = (sel(Ax), Con)$ *for any given problem* $P = (Ax, Con)$*.*

A selection function *sel* is called *complete* if $Ax \models Con \Leftrightarrow sel(Ax) \models Con$ for any problem $P = (Ax, Con)$. With respect to the equivalent satisfiability problem P^{sat} a complete selection function ensures that P^{sat} and $sel(P^{sat})$ are equisatisfiable.

E-KRHyper features both complete and incomplete selection functions.

12.2.1 Complete Selection

A complete selection function may only remove axioms which are irrelevant for a proof of the conjecture. Regarding the equivalent satisfiability problems a complete selection would ideally select only the smallest unsatisfiable subset of a given problem. For the LogAnswer test problems there is an enormous potential for savings here - recall that the test set problems are usually solved with less than 0.1% of their clauses (see Section 9.2). However, in most circumstances this subset is only known after the proof.

This has not stopped attempts to achieve the ideal: In the CASC of 2008 the Vampire prover featured a curious two-phase solution in which it first converted the problem into Prolog and let Prolog try to find a proof within some time limit. If Prolog succeeded, then the clauses involved in the proof were handed over to the actual Vampire prover, which derived a bottom-up proof for the reduced clause set. The solution was ideal in that the unsatisfiable subset got selected, but naturally this first required a proof just for the selection. The hope likely was that the goal-directed top-down approach of Prolog could be advantageous for large problems, while the second proof by Vampire was merely required for a valid competition result. This method was dropped in later years.

In practice such a proof-based approach is undesirable, in particular in the LogAnswer use case which does not allow the generous CASC time limits. The selection must be quick, there is no time to allow it to perform much reasoning. E-KRHyper implements one complete axiom selection scheme, internally referred to as *partitioning*. As selecting the unsatisfiable subset is unattainable in practice, our approach instead deselects clauses which are guaranteed not to be in the unsatisfiable subset. This approach is considerably weaker, but much faster. The partitioning operates on a set of clauses $\mathcal{C}$ and extracts a subset $sel(\mathcal{C})$ that is equisatisfiable to $\mathcal{C}$. It is primarily intended for equality-free problems under the hyper tableaux calculus, although with modifications it can be used with the E-hyper tableaux calculus and problems with equality. The

selection function utilizes the fact that the predicate symbols of the negative literals of a clause selected in a hyper extension step also occur in positive branch literals, a direct corollary of the hyper condition. We introduce two notions about predicate symbols, relevance and derivability. First, let $neg(C)$ denote the set of predicate symbols of the negative literals of a clause C and let $pos(C)$ analogously denote the predicate symbols of the positive literals.

Definition 12.2 (Relevance)**.** *Given a clause set $\mathcal{C}$ and two predicate symbols P and Q. P* is relevant for *Q if*

1. *$P \in neg(C)$ and $Q \in pos(C)$ for some clause $C \in \mathcal{C}$, or*
2. *$P \in neg(C)$ and $R \in pos(C)$ for some clause $C \in \mathcal{C}$ and R is relevant for Q.*

Then P is relevant *if*

1. *$P \in neg(C)$ for some negative clause $C \in \mathcal{C}$, or*
2. *P is relevant for some relevant predicate symbol Q.*

Intuitively *P is relevant for Q* since Q depends on P in as much that positive branch literals with P may become involved in a hyper extension step that leads (directly or indirectly) to the derivation of positive branch literals with Q. The more general notion of P being *relevant* indicates that literals with P may become involved in closing a branch, directly or indirectly.

Definition 12.3 (Derivability)**.** *A predicate symbol P is* derivable *if $P \in pos(C)$ for some clause $C \in \mathcal{C}$ and every $Q \in neg(C)$ is derivable.*

Note that this recursive definition implies that all predicate symbols of positive clauses are immediately derivable. Intuitively the derivability of P signifies that positive branch literals with P may occur during a derivation for $\mathcal{C}$.

Then a clause $C \in \mathcal{C}$ is selected ($C \in sel(\mathcal{C})$) if

1. every $P \in neg(C)$ is derivable, and
2. (a) C is negative, or
   (b) some $Q \in pos(C)$ is relevant.

As an example consider the following clause set $\mathcal{C}$:

(1) $\leftarrow p(x), q(y)$

(2) $p(x) \leftarrow r(x)$

(3) $r(x) \leftarrow p(y), u(x)$

(4) $q(x) \leftarrow s(x)$

(5) $t(x) \leftarrow p(x)$

(6) $p(a) \leftarrow$

(7) $s(b) \leftarrow$

Here the relevant predicate symbols are $\{p/1, q/1, r/1, s/1, u/1\}$ and the derivable predicate symbols are $\{p/1, q/1, s/1, t/1, v/1\}$. The set of selected clauses then is $sel(\mathcal{C}) = \{(1), (4), (6), (7)\}$. (2) is not selected because $r/1$ is not derivable and (3) is not selected because $u/1$ is not derivable. (5) fails the selection criteria in that $t/1$ is not relevant.

Proposition 12.1. *The selection function sel used by the partitioning in E-KRHyper is complete.*

Proof. Let $\mathcal{C}$ be any reasoning problem, and assume without loss of generality that $\mathcal{C}$ is represented as a satisfiability problem in clause normal form. We show that P is equisatisfiable to $sel(\mathcal{C})$. This requires to show that:

1) if $\mathcal{C}$ is satisfiable, then $sel(\mathcal{C})$ is satisfiable,

2) if $\mathcal{C}$ is unsatisfiable, then $sel(\mathcal{C})$ is unsatisfiable,

3) if $sel(\mathcal{C})$ is satisfiable, then $\mathcal{C}$ is satisfiable,

4) if $sel(\mathcal{C})$ is unsatisfiable, then $\mathcal{C}$ is unsatisfiable.

1) follows from $sel(\mathcal{C}) \subseteq \mathcal{C}$ and the compactness of first-order logic. 2) is proven by refutation. Assume $\mathcal{C}$ to be unsatisfiable and $sel(\mathcal{C})$ to be satisfiable. Then there is hyper tableaux refutation and a closed hyper tableau for $\mathcal{C}$. From the refutation we can extract for each branch a minimal finite sequence $\mathcal{I} = I_1, \ldots, I_n$ of hyper extension step inferences, where every $I_i = (B_i, C_i, D_i)$ for $1 \leq i \leq n$ is represented as a tuple consisting of the extending clause C_i, the set B_i of the positive branch literals serving as premises and the set D_i of the resulting positive branch literals, and where for every $L \in B_i$ there is an $I_j = (B_j, C_j, D_j)$ with $1 \leq j < i$ and $L \in D_j$, and where D_i either is empty or for some $K \in D_i$ there is an $I_k = (B_k, C_k, D_k)$ with $i < k \leq n$ and $K \in B_k$. That is, every premise literal must have been a result literal of an earlier hyper extension step, and every result literal set either closes the branch or contains a literal that serves as a premise in a later hyper extension step.

Also, the following properties hold:

***Rel*$_1$:** Every result set D_i is either empty or contains at least one relevant predicate symbol.

***Rel*$_2$:** Every premise set B_i contains only relevant predicate symbols.

This is easily shown: If $i = n$, then D_i is empty and Rel_1 holds. C_i is a negative clause and all literals in B_i unify with the atoms of this negative clause, which makes the predicate symbols in B_i relevant, thus Rel_2 holds. For $i < n$ we prove the properties by contradiction. Assume $I_o = (B_o, C_o, D_o)$ with $o < n$ to be the last inference tuple in $\mathcal{I}$ for which Rel_1 or Rel_2 does not hold, i.e. for all later I_r in $\mathcal{I}$ with $o < r \leq n$ both properties hold. D_o is not empty, because otherwise C_o would be a negative clause and I_o would already have closed the branch, so $\mathcal{I}$ would not be minimal. This means that some literal $L \in D_o$ must also occur in some later premise set B_r with $o < r \leq n$. But then L has a relevant predicate, as property Rel_2 holds for B_r and I_r, and thus Rel_1 holds for D_o and I_o. Since L is an instance of a positive literal of C_o, its relevant predicate symbol is contained in $pos(C_o)$, and hence all $neg(C_o)$ are relevant as

well. As all literals in B_o unify with the atoms of the negative literals of C_o, all predicate symbols in B_o are relevant, so Rel_2 holds for B_o and I_o. Therefore Rel_1 and Rel_2 both hold for I_o, a clear contradiction.

Furthermore this property holds:

***Der*$_1$:** Every premise set B_i contains only derivable predicate symbols.

This is shown in a similar manner as for Rel_1 and Rel_2. Assume $I_o = (B_o, C_o, D_o)$ to be the first inference tuple in $\mathcal{I}$ for which Der_1 does not hold, i.e. for all earlier I_r in $\mathcal{I}$ with $r < o$ the property Der_1 holds. If C_o is a positive clause, then B_o is empty and Der_1 trivially holds. Assume therefore that C_o contains negative literals and thus B_o is not empty. Then each literal $L \in B_o$ must also occur in some earlier result set D_r with $r < o$. As Der_1 holds for B_r and I_r, it follows that all predicate symbols in $neg(C_r)$ are derivable, since the atoms of all the negative literals of C_r must unify with the literals in B_r. Then also all predicate symbols in $pos(C_r)$ are derivable, and so are all predicate symbols in D_r, since all literals in D_r are instances of positive literals of C_r. But then the predicate symbol of L must be derivable as well. As this is the case for each $L \in B_o$, the property Der_1 holds for B_o and I_o, a clear contradiction.

Now, since $sel(\mathcal{C})$ is satisfiable, there must be some branch in the closed hyper tableau for $\mathcal{C}$ that cannot be closed when using only the clauses from $sel(\mathcal{C})$. The inference sequence $\mathcal{I}$ of that branch must then contain some earliest failing inference $I_s = (B_s, C_s, D_s)$ where $1 \leq s \leq n$ and $C_s \in \mathcal{C}$ and $C_s \notin sel(\mathcal{C})$, such that there is no earlier failing inference $I_t = (B_t, C_t, D_t)$ with $1 \leq t < s$ and $C_t \in \mathcal{C}$ and $C_t \notin sel(\mathcal{C})$. Since every P in B_s is derivable as per Der_1 and all atoms of the negative literals of C_s unify with literals from B_s, every $Q \in neg(C_s)$ is derivable, so C_s meets the first selection criterion. As per Rel_1, either D_s is empty and then C_s is negative and thus meeting selection criterion 2.(a), or D_s contains a relevant predicate symbol, which means that $pos(C_s)$ contains this relevant symbol as well, thus meeting selection criterion 2.(b). Thus $C_s \in sel(\mathcal{C})$, a clear contradiction to $C_s \notin sel(\mathcal{C})$.

For 3) assume $sel(\mathcal{C})$ to be satisfiable and $\mathcal{C}$ to be unsatisfiable. Then again there is a hyper tableaux refutation for $\mathcal{C}$, leading to a contradiction analogous to 2).

Finally, 4) like 1) follows from $sel(\mathcal{C}) \subseteq \mathcal{C}$ and the compactness of first-order logic. □

Due to its completeness this selection method can be used for theorem proving in general, although one has to be aware that it is only refutationally complete: A model found for a satisfiable $sel(\mathcal{C})$ may not be sufficient as a model for $\mathcal{C}$. E-KRHyper employs the partitioning for all reasoning problems in LogAnswer. Here it has the advantage of the compact MultiNet predicate symbol inventory. As multi-literal clauses only occur in the background knowledge base shared between all problems, the dependencies between the predicate symbols can be computed in advance as soon as this shared background knowledge base has been loaded. A dependency graph is used for this. Once a specific question has been loaded together with an answer candidate, the FOL query as the only negative clause determines which predicate symbols are relevant, while predicate symbols occurring in positive clauses determine derivability. Since there are only 86 different predicates and the predicate representations are shared between clause data structures, the clauses not matching the selection criteria are

quickly deactivated. On average the partitioning removes 20% of the clauses from a LogAnswer reasoning problem, which reduces the proof time by 10%. This is a minor saving, but it is achieved with minimal computational effort and preserves refutational completeness.

Our approach has some similarity to the selection method based on *fully matched sets* [PY03], which links two clauses if they have a pair of complementary literals with unifiable atoms. As that method uses actual unification instead of correspondence between predicate symbols, it is more difficult to compute and not easily adapted to LogAnswer. Unlike our partitioning this approach cannot profit much from a static shared knowledge base and a stable predicate symbol inventory. For each question and answer candidate it would instead have to test hundreds of passage-representing literals for unification with the knowledge base, just to perform the selection. Also, it does not use any notion of derivability to remove clauses that, while relevant for the query, will nevertheless never become involved in a bottom-up proof.

The disadvantages of our partitioning should be mentioned. With a growing number of predicate symbols the computation of dependencies becomes increasingly difficult. Even on smaller TPTP subsets of Cyc the computation of the dependency graph is impracticable. Also, the method as described is not complete for problems with equality. For equality-free problems the reasoning is predicate-driven in that there is always some matching of predicate symbols between inference premises. However, in problems with equality the superposition-based inferences do not have this limit, as equations can rewrite other literals regardless of their predicates. This can be remedied by always regarding the equality predicate as relevant and by modifying the aspect of derivability so that clauses also count as positive if all their negative literals are equations. However, given that even the stricter original partitioning selection function only deselects a modest number of clauses, such an adaptation leads to virtually no savings when dealing with equational problems. When equations enter the game, it seems preferable to forgo completeness, which leads us to incomplete selection methods.

12.2.2 Incomplete Selection

Incomplete selection functions select a subset of a given problem, but since they do not guarantee to preserve completeness, they can use heuristics to pick much smaller axiom sets. The lack of completeness means that the results must be handled with care: While an unsatisfiable set $sel(P)$ also proves the original problem P unsatisfiable via compactness, a satisfiable $sel(P)$ allows no conclusions regarding the satisfiability of P; it may indeed be satisfiable, or it is unsatisfiable and the selection function just missed picking some axiom critical for the contradiction. Incomplete methods lend themselves to iterative approaches. They can severely reduce a large problem to a comparatively tiny set of axioms, and doing so increases the chances that the prover finishes its derivation fast - it usually succeeds fast or it fails fast, rather than computing indefinitely. After an early failure a new attempt can be started with a different selection method, a process that is repeated with increasingly larger selections until a proof is found or the full problem set is used.

E-KRHyper implements two incomplete selection methods, the SInE algorithm and an approach we devised to address some weaknesses of SInE.

The SInE Selection

The *SInE* (SUMO Inference Engine) algorithm [HV11] was developed as an axiom selection method for very large reasoning problems. Originally implemented as a preprocessor, it has since become integrated in a number of theorem provers. The advantages of SInE are a consequence of its selection criteria which are very simple to compute, yet which result in very small subsets that are frequently unsatisfiable.

We give a short overview of the SInE selection algorithm. Let $sym(F)$ denote the set of function and predicate symbols (except equality) occurring in a formula (or clause) F. Given a set of axioms $\mathcal{A}$, let $occ(s)$ denote the number of axioms in $\mathcal{A}$ which contain the symbol s. For any axiom $A \in \mathcal{A}$, let $r(A)$ denote the set of the rarest symbols in A. That is, $r(A) \subseteq sym(A)$ and for any $s \in r(A)$, $occ(s) \leq occ(s')$ for any symbol $s' \in sym(A)$.

The SInE-algorithm is goal-oriented, thus we assume some goal G, usually in the form of a conjectural formula or clause. Then first select every $A \in \mathcal{A}$ with $\exists s \in r(A)$ such that $s \in sym(G)$. After that, select every unselected $A \in \mathcal{A}$ with $\exists s \in r(A)$ such that $s \in sym(A')$ for some already selected $A' \in \mathcal{A}$. The second step is repeated until a fixed-point is reached.

The algorithm can be parameterized, for example by restricting the repetitions of the second step to n iterations, or by introducing a tolerance factor that allows the selection of an axiom A by a symbol that is not quite the rarest within A.

The basic idea behind the algorithm is that if a symbol s is rare, then it is more important within an axiom than the other symbols of that axiom, and an axiom containing s is important for defining s. The algorithm thus tries to capture semantic aspects purely by syntactical and numerical means, with surprisingly successful results - we refer to [HV11] for an evaluation of the algorithm, but suffice it to write that in CASC the SInE algorithm has become the dominant axiom selection method, with all participants in the LTB division of 2011 using some form of SInE.

The E-KRHyper Selection

An advantage of SInE is that it can operate on axioms that have been subjected to hardly any processing yet. It does not need to transform formulas into clauses, and comparatively simple data structures are enough for the computation. Within a theorem prover a clause is usually a fairly heavyweight structure; apart from the literals it typically contains pointers to indexing trees and information about clause features like whether it is ground, Horn, and so on. For SInE it is not necessary to provide these structures, it can operate on axiom representations that are little more than the strings read from the problem specification. Therefore it can perform selections on massive knowledge bases like Cyc, since the selection can be computed before the prover starts to construct clause representations and to store them in indexing trees.

However, the lack of any logical processing leads to a flaw that can result in obviously satisfiable and hence useless selections. We encountered this when we tested the early SInE 0.3 system in an initial experiment with the LogAnswer problem test set [8]. Here SInE 0.3, consisting of a SInE algorithm preprocessor and the prover E, solved 72.6% of the problems, whereas E on its own solved

92.2%. The updated testing in this dissertation (see Section 9.3) shows a similar discrepancy for the current SInE 0.4, which now solved more problems than before, but still less than E and E-KRHyper, and it was markedly slower than SInE 0.3, E and E-KRHyper.[1] Our investigation showed that the difficulties of SInE with the LogAnswer problems are mostly a result of the following situation which frequently occurs in some knowledge bases: Large numbers of facts are stored as ground units with a common predicate, and a query that requires one of these facts triggers a search via a non-ground literal of the same predicate. The predicate is very common in the knowledge base, which means that SInE avoids selecting any of its units, thereby failing to select an unsatisfiable subset.

For example, consider the following query Q and the small knowledge base consisting of the unit clauses $C_1, \ldots, C_7$:

Q**:** $\leftarrow \mathit{predator}(x), \mathit{mammal}(x)$

C_1**:** $\mathit{predator}(\mathit{wolf}) \leftarrow$

C_2**:** $\mathit{predator}(\mathit{eagle}) \leftarrow$

C_3**:** $\mathit{mammal}(\mathit{wolf}) \leftarrow$

C_4**:** $\mathit{mammal}(\mathit{cow}) \leftarrow$

C_5**:** $\mathit{mammal}(\mathit{sheep}) \leftarrow$

C_6**:** $\mathit{bird}(\mathit{eagle}) \leftarrow$

C_7**:** $\mathit{bird}(\mathit{hummingbird}) \leftarrow$

The SInE algorithm will select axioms via the query predicate symbols *predator*/1 and *mammal*/1. For *predator*/1 it selects C_1 and C_2, as *predator*/1 is among the rarest symbols there - it occurs twice in the entire knowledge base. However, for *mammal*/1 no axioms are selected, as the *mammal*/1 symbol occurs three times in the knowledge base and is thus more common than the animal names in the unit clauses. Instead the selection of C_2 triggers the selection of C_6 via *eagle*, as both *eagle* and *bird*/1 are equally rare. Without any of C_3, C_4 and C_5 it is obviously impossible to prove the goal Q.

A tolerance factor would help in this case, but in real-world knowledge bases the same predicate may be used to store thousands of facts, making their predicate so common as to render these facts inaccessible to SInE, unless the tolerance is increased so far that nearly the entire knowledge base is selected.

E-KRHyper therefore features another incomplete selection method that we based on SInE in an attempt to remedy this shortcoming. Using the fast clausifier of E-KRHyper (see Section 7.4.4) our method is intended to operate on clauses in CNF. Let $sym_f^+(C)$ denote the set of function symbols occurring in positive literals of a clause C, and let $sym_p^+(C)$ be the predicate symbols (except equality) occurring in the positive literals. Let $sym_f^-(C)$ and $sym_p^-(C)$ be the analogous sets for the symbols from negative literals. Given a set of clauses $\mathcal{C}$, let $occ(s)$ denote the number of clauses in $\mathcal{C}$ which contain the symbol s. For any clause $C \in \mathcal{C}$, let $r_f^+(C)$ denote the set of the rarest positively occurring

[1] The slowdown between SInE 0.3 and SInE 0.4 can be explained by differences in their iterative approach, with the 0.4 version allowing larger subsets by higher tolerance settings, thus succeeding more often, but requiring more time.

function symbols in C. That is, $r_f^+(C) \subseteq sym_f^+(C)$ and for any $s \in r_f^+(C)$, $occ(s) \leq occ(s')$ for any symbol $s' \in sym_f^+(C)$. Analogously we define $r_p^+(C)$ for the rarest positive predicate symbols and $r_f^-(C)$ and $r_p^-(C)$ for the rarest symbols in negative literals.

Then given a goal clause G, first select:

- every $C \in \mathcal{C}$ with $\exists s \in r_f^+(C)$ such that $s \in sym_f^-(G)$,
- every $C \in \mathcal{C}$ with $\exists s \in r_f^-(C)$ such that $s \in sym_f^+(G)$,
- every $C \in \mathcal{C}$ with $\exists s \in r_p^+(C)$ such that $s \in sym_p^-(G)$,
- every $C \in \mathcal{C}$ with $\exists s \in r_p^-(C)$ such that $s \in sym_p^+(G)$.

After that select:

- every $C \in \mathcal{C}$ with $\exists s \in r_f^+(C)$ such that $s \in sym_f^-(C')$ for some already selected $C' \in \mathcal{C}$,
- every $C \in \mathcal{C}$ with $\exists s \in r_f^-(C)$ such that $s \in sym_f^+(C')$ for some already selected $C' \in \mathcal{C}$,
- every $C \in \mathcal{C}$ with $\exists s \in r_p^+(C)$ such that $s \in sym_p^-(C')$ for some already selected $C' \in \mathcal{C}$,
- every $C \in \mathcal{C}$ with $\exists s \in r_p^-(C)$ such that $s \in sym_p^+(C')$ for some already selected $C' \in \mathcal{C}$.

The second step is repeated until a fixed-point is reached.

By distinguishing between function and predicate symbols, a clause can be selected by a predicate symbol that is more common than its function symbols or vice versa. In the example above the E-KRHyper axiom selection method would thus first select C_1 and C_2 for *predator*/1 and C_3, C_4 and C_5 for *mammal*/1. Unlike with SInE there is no selection of C_6 triggered by C_2 via *eagle*, because this symbol occurs only in positive literals in both clauses, while our selection method requires complementary occurrences. With the selection of C_1 and C_3 the query Q can now be proven.

In general our method selects more clauses than SInE does, even though the distinction between positive and negative occurrences helps to avoid some unnecessary selections made by SInE.

12.3 Evaluation

In this section we evaluate the different selection methods presented in this chapter. In the first test series we compare the complete partitioning selection from Section 12.2.1 with the incomplete methods from Section 12.2.2 on our test set of 1,805 LogAnswer problems. As usual the test hardware features an Intel Q9950 CPU with 2.83 GHz. The time limit per problem is set to 60 seconds, and the memory was limited to 1 GB. To make proper use of the axiom selection algorithms, the prover was kept in continuous operation while processing the 1,805 problems with one selection method; it was restarted only for the next

selection type	solved	$\bar{t}$	$\tilde{t}$
no selection	1,728 (95.7%)	0.99 s	0.4 s
partitioning	1,728 (95.7%)	0.89 s	0.3 s
SInE	745 (41.3%)	0.01 s	0.01 s
E-KRHyper	861 (46.6%)	0.06 s	0.05 s

Table 12.1: Evaluation results for E-KRHyper with selection methods on LogAnswer problem test set

run with the next selection method. The weight setting in E-KRhyper was optimized for LogAnswer problems as in Section 9.3.

The results are summarized in Table 12.1. The first column specifies the selection method for the particular run, and the second column states the number of problems solved. The columns $\bar{t}$ and $\tilde{t}$ indicate the average and the median solving time respectively. As a baseline the result of the evaluation of continuous operation with input reuse (see Section 11.5) is included, because it used the same test conditions, but without any selection algorithm. Compared to this, usage of the complete partitioning algorithm reduced the average solving time by 10.1% and the median solving time by 25%. The number of problems solved remained the same at 1,728, which is 95.7% of the total test set.

The SInE selection achieved by far the best solving times, with 0.01 seconds both on average and as a median. However, only 41.3% of the problems were solved, and these are a subset of the problems solved with complete partitioning. If we regard only this subset, then the complete selection method achieved an average solving time of 0.76 seconds and a median solving time of 0.1 seconds. Moreover, in practice we cannot disregard the unsolved problems. Of the 1,060 problems not solved after the SInE selection, 425 problems (23.5% of the total test set) resulted in a false satisfiability result. These were relatively benign, as on average these results were found after 0.01 seconds. The 635 problems remaining unsolved after SInE (35.2% of the total set) however kept the prover occupied until some limit (time, memory or term size) was reached. In practice all these failed problems would then require a second evaluation by a more complete method.

Also, one has to keep in mind that the test set consists of problems that were solvable in their original form, and which usually are solvable in their FOL representation as well - the intention behind this set is to test how well a system does in solving the problems. In actual LogAnswer usage on the other hand only a minority of the problems presented to the prover can be solved even within a more generous time limit: In the CLEF competition of 2010 only 7% of the proof attempts resulted in a proof, and this was with E-KRHyper using the complete partitioning and a time limit of several seconds. Thus if SInE with its 41.3% success rate on solvable problems were to be used as the primary selection method in all proof attempts, then the system would succeed only in about 3% of all attempted proofs, requiring a fallback to the complete method in all other cases. With the complete method therefore being used eventually in 97% of the proof attempts and the additional overhead involved in resetting the prover after a failed SInE-attempt, it is preferable to use the complete method right away.

selection type	solved	exclusive
no selection	226 (15%)	10 (0.6%)
partitioning	199 (13.2%)	7 (0.5%)
SInE	240 (15.9%)	114 (7.6%)
E-KRHyper	172 (11.4%)	13 (0.9%)

Table 12.2: Evaluation results for E-KRHyper with selection methods on LTB problem test set

This changes when a QA system utilizes parallel computing: By employing two prover instances, one can process a problem with a complete selection while the other uses an incomplete method. If the incomplete method succeeds, the complete attempt can be aborted. The current LogAnswer does not support parallel computing, so this must be regarded as future work.

The evaluation result for our alternative SInE-based incomplete selection method in E-KRHyper shows an increase both in the number of problems solved and in the solving time. While it thus shows some success in addressing the aforementioned flaw of SInE (see Section 12.2.2), for the LogAnswer problems it can merely be seen as a compromise between SInE and the complete selection, the solving rate still being too low to justify the usage in a non-parallel LogAnswer.

Incomplete selection methods show their advantage when dealing with very large ontologies. We tested the complete partitioning and both incomplete methods with E-KRHyper on the 1,509 TPTP problems eligible for the LTB division in the CASC of 2011. These are very large problems derived from work with the Cyc and SUMO ontologies, as well as large mathematical problems from the projects *Isabelle*[2] [Pau89] and *Mizar*[3] [NK09]. As these are problems from the TPTP, we used the standard TPTP testing time limit of 300 seconds. The memory remained limited to 1 GB, and the hardware was our standard system as above.

The results are summarized in Table 12.2. The E-KRHyper results for these problems when using no selection algorithm are extracted from the normal TPTP testing, see Section 7.5. Some interesting observations can be made. When E-KRHyper used the complete partitioning, it actually solved fewer problems than when working with the full problems. This is because the computation of the dependencies takes considerable time on large problems while offering only small reductions, and thus the prover ends up having less time to perform almost the same amount of reasoning work. Another observation is that none of the four sets of solved problems is fully subsumed by one of the others, as can be seen in the *exclusive* column stating the number of problems that were only solved with the respective method. 7.6% of the test set were only solved by employing the SInE selection, a significant edge in this criterion, and obviously there is much greater overlap between the problem sets solved by the other three methods.

The four approaches combined solved 388 problems, representing the maximum E-KRHyper can achieve on this problem set. Due to the overlap between

[2] `http://www.cl.cam.ac.uk/research/hvg/Isabelle`
[3] `http://mizar.org`

the sets of problems solved by the different approaches, combining fewer of the selection methods can approximate the maximum result with less processing. When combining the two with the most solutions, SInE and the full problems without selection, 368 problems were solved, 94.8% of the maximum. The two incomplete methods together solved 304 problems, 78.4% of the maximum. While solving less than the previous combination, substituting the E-KRHyper selection for the full problem usage has advantages regarding the processing time. Without selection the average solution took 14.8 seconds, while the incomplete E-KRHyper selection required 5.2 seconds. Considering the 133 problems only that both approaches solved, then without selection it took on average 8.4 seconds for each solved problem whereas with the E-KRHyper selection it took 2.8 seconds. Under stringent time constraints it is therefore appropriate to combine the incomplete selection methods, by first applying one, and then using the other if the first one fails. Under more generous time limits it makes sense to first use SInE and then fall back upon the full problems in a similar manner.

Chapter 13

Robustness

By the time the logical representations of the question, the text passage and the background knowledge are loaded into the theorem prover E-KRHyper, the data has passed through multiple stages and translations, each of which can introduce errors:

1. The textual sources, in our case Wikipedia, may contain incorrect information.

2. The parser which reads the textual sources can fail at correctly resolving ambiguous or complex statements, and it may become derailed by syntactic problems like spelling mistakes or peculiarities in the layout, for example embedded figures and tables.

3. The tools for the automatic translation into semantic networks do not yet cover all aspects of MultiNet, so some information from the text may not make it into the MultiNet representation despite correct parsing.

4. Each network fragment and candidate passage obtained by the information retrieval phase of LogAnswer is based on a single sentence from the textual sources. Important information from the original context in the textual sources may hence be missing in the answer candidate.

5. The expressivity of MultiNet exceeds the means of first-order logic, so information from the semantic network representation may not make it into the FOL representation.

Thus the logical representation of a text passage is bound to have omissions. Combined with the wealth of synonyms and paraphrases it is therefore unlikely that a passage representation matches a query so exactly that a proof is possible - its precision makes deduction too brittle, see Section 5.2.3. Indeed, of the question and candidate passage representations evaluated in CLEF 2010, only 0.7% were immediately provable with E-KRHyper. The 7% proof rate mentioned in Section 12.3 could only be achieved with the robustness enhancement of *relaxation*, as described in this chapter.

13.1 Relaxation

Generally relaxation refers to a loosening of constraints in an attempt to simplify a problem, in the hope of finding a solution to the simpler problem that also applies to the original. In the case of LogAnswer relaxation refers to a technique that simplifies a given question, making it less specific. The simpler question may then be easier to answer, and the answer may still be relevant for the original question. For example, consider this pair of question and candidate passage:

Q**:** *"What is the tallest mountain in the world?"*

C**:** *"Mount Everest is the tallest mountain."*

Technically the candidate passage does not contain enough information to answer the question, as it contains no reference to *"the world"*. A strict QA system would have to give up here. With relaxation we can make the question less specific:

$Q^{\mathbf{rel}}$**:** *"What is the tallest mountain?"*

The passage contains sufficient information to answer this question. In this particular case the answer *"Mount Everest"* is also the correct answer to the original question. Naturally relaxation contains some risk that the simplified question becomes too unspecific, making the answer irrelevant for the original question. For example, the question *"What is a mountain in the world?'* is also a relaxed version of Q, yet it would be mere coincidence if an answer were relevant for Q.

Relaxation in LogAnswer is carried out as a cooperation between the theorem prover E-KRHyper and the robust logic-based processing module (see Section 8.4) [15]. When E-KRHyper fails to find a proof within a specified time limit, then the FOL question representation is shortened by one literal and the prover is restarted with this relaxed query. As a FOL query is a negative clause Q, the relaxed query Q^{rel} is a subset of Q. With fewer query literals to refute, finding a refutation should be easier. As shown by the example above, some thought must be put into the choice of which literal to skip. Obviously the best choice is a literal that is not provable, but a query clause may have several provable subsets - indeed every literal may be provable, but not all of them simultaneously under a common substitution. We wish to relax the query as little as possible, so it is desirable to preserve the largest provable subset and to skip only literals from the remainder of the query clause. As a query clause of length n has $2^n - 2$ proper non-empty subsets, it is unfeasible to test all of them in addition to the full query, but as described below we do test several subsets to achieve an approximate result that is better than randomly skipping some literal.

Towards this purpose E-KRHyper gathers information about its progress in proving the query clause during the derivation. If the time limit is reached without a proof, then this information is used by the robust logic-based processing module to select an appropriate literal for skipping, and then E-KRHyper is restarted with the relaxed query. This information is obtained whenever E-KRHyper tries to apply a hyper extension step to a query clause $Q = \neg Q_1 \vee \cdots \vee \neg Q_n$. In such an attempt the prover tries to unify the query atoms with

fresh variants $B_1, \ldots, B_n$ of positive literal nodes (unit clauses) from the current hyper tableau branch. It iterates over Q from left to right, starting with $\neg Q_1$, and extends a unifying substitution in every step such that starting with the empty substitution σ_0, for any k with $1 \leq k \leq n$ there is a substitution σ_k with $Q_i\sigma_k = B_i$ $(1 \leq i \leq k)$. If one query atom Q_l fails to find a matching partner in the branch which would allow an extension of the current substitution σ_{l-1} to σ_l, then the remaining literals $\neg Q_{l+1}, \ldots, \neg Q_n$ are not tested under σ_{l-1}. Instead E-KRHyper stores a *partial result*, which is a triple of the form:

$$(\{Q_1, \ldots, Q_{l-1}\}, \sigma_{l-1}, \{Q_l, \ldots, Q_n\})$$

It consists of the list of successfully unified query atoms, their unifying substitution, and the list of query atoms that were not unified, the first of which being the one that failed, whereas the remaining were not tested at all.

The linear left to right evaluation could lead to an early failing literal preventing a large subset of Q from ever being tested. To avoid this, E-KRHyper performs the same evaluation for the following permutations of Q:

$$\neg Q_2 \vee \neg Q_3 \vee \cdots \vee \neg Q_n \vee \neg Q_1$$

$$\neg Q_3 \vee \neg Q_4 \vee \cdots \vee \neg Q_n \vee \neg Q_1 \vee \neg Q_2$$

$$\vdots$$

$$\neg Q_n \vee \neg Q_1 \vee \cdots \vee \neg Q_{n-1}$$

While this will not cover all subsets, it ensures that every literal is tested at least once, thereby allowing the detection of complete failures where no query literals can be proven within the time limit.

The default setting of E-KRHyper in LogAnswer has been to gather all partial results during the derivation. This method has been used in the CLEF competition participations of LogAnswer. An experimental setting enables E-KRHyper to discard partial results which become obsolete due to better partial results. This is the case when the set of proven literals in a partial result R is a subset of the proven literals of a partial result R', a situation that can occur when some unproven literal in R is proven later by some newly derived fact. However, this "subsumption" of partial results is merely an optimization, as the robust logic-based processing module still needs to rank and evaluate the partial results according to criteria that are beyond the means of E-KRHyper. Here the literal to skip is determined by the partial result with the most proven literals, and when several partial results are tied, preference is given to those that bind the *FOCUS* variable to a constant term that represents a MultiNet node which is likely to be a suitable answer.

As an example consider the following question and candidate passage:

Q**:** *"Rudy Giuliani war Bürgermeister welcher US-Stadt?"*[1]

C**:** *"Hinter der Anklage stand der spätere Bürgermeister von New York, Rudolph Giuliani."*[2]

[1] *"Rudy Giuliani was the mayor of which city in the USA?"*
[2] *"Responsible for the charges was the future mayor of New York, Rudolph Giuliani."*

The question is represented as the following negative clause:

$$
\begin{aligned}
Q = \quad & \neg attr(X_1, X_2) \vee \neg attr(X_1, X_3) \vee \neg sub(X_2, nachname.1.1) \\
& \vee \neg val(X_2, giuliani.0) \vee \neg sub(X_3, vorname.1.1) \\
& \vee \neg val(X_3, rudy.0) \vee \neg sub(X_1, buergermeister.1.1) \\
& \vee \neg attch(FOCUS, X_1) \vee \neg sub(FOCUS, usstadt.1.1)
\end{aligned}
$$

The candidate passage has this FOL representation (simplified for legibility):

$$
\begin{aligned}
C = \quad & hinter(c221, c210) \wedge sub(c220, nachname.1.1) \wedge val(c220, giuliani.0) \\
& \wedge sub(c219, vorname.1.1) \wedge val(c219, rudolph.0) \wedge prop(c218, spaet.1.1) \\
& \wedge attr(c218, c220) \wedge attr(c218, c219) \wedge sub(c218, buergermeister.1.1) \\
& \wedge val(c216, new\text{-}york.0) \wedge sub(c216, name.1.1) \wedge sub(c215, stadt.1.1) \\
& \wedge attch(c215, c218) \wedge attr(c215, c216) \wedge subs(c211, stehen.1.1) \\
& \wedge loc(c211, c221) \wedge scar(c211, c218) \wedge sub(c210, anklage.1.1)
\end{aligned}
$$

The initial proof attempt does not succeed. The best partial result instantiates the first five query literals, but E-KRHyper fails at $\neg val(X_3, rudy.0)$, because the knowledge base contains no information that *"Rudy"* is a short form of *"Rudolph"*:

$$
\begin{aligned}
& (\{\, attr(X_1, X_2), attr(X_1, X_3), sub(X_2, nachname.1.1), val(X_2, giuliani.0), \\
& \quad sub(X_3, vorname.1.1) \,\}, \\
& \sigma = \{X_1 \leftarrow c218, X_2 \leftarrow c220, X_3 \leftarrow c219\}, \\
& \{\, val(X_3, rudy.0), sub(X_1, buergermeister.1.1), attch(FOCUS, X_1), \\
& \quad sub(FOCUS, usstadt.1.1) \,\})
\end{aligned}
$$

Thus the failing literal is skipped, resulting in the relaxed query:

$$
\begin{aligned}
Q^{rel} = \quad & \neg attr(X_1, X_2) \vee \neg attr(X_1, X_3) \vee \neg sub(X_2, nachname.1.1) \\
& \vee \neg val(X_2, giuliani.0) \vee \neg sub(X_3, vorname.1.1) \\
& \vee \neg sub(X_1, buergermeister.1.1) \vee \neg attch(FOCUS, X_1) \\
& \vee \neg sub(FOCUS, usstadt.1.1)
\end{aligned}
$$

Accordingly E-KRHyper starts a new proof attempt with Q^{rel}. Most of the remaining literals can now be proven, apart from the last one, yielding this partial result:

$$
\begin{aligned}
&(\{\ \mathit{attr}(X_1, X_2), \mathit{attr}(X_1, X_3), \mathit{sub}(X_2, \mathit{nachname.1.1}), \mathit{val}(X_2, \mathit{giuliani.0}),\\
&\quad \mathit{sub}(X_3, \mathit{vorname.1.1}), \mathit{sub}(X_1, \mathit{buergermeister.1.1}), \mathit{attch}(\mathit{FOCUS}, X_1)\ \}\ ,\\
&\sigma = \{X_1 \leftarrow \mathit{c218}, X_2 \leftarrow \mathit{c220}, X_3 \leftarrow \mathit{c219}, \mathit{FOCUS} \leftarrow \mathit{c215}\}\ ,\\
&\{\ \mathit{sub}(\mathit{FOCUS}, \mathit{usstadt.1.1})\ \})
\end{aligned}
$$

While the knowledge base contains data about New York being a city, the crucial fact that it is a city in the USA (*usstadt.1.1*) is missing. The literal is skipped as well, but since there are no further query literals to prove, no further proof attempt must be started. The *FOCUS* variable has been bound to the constant *c215*, and from the passage representation the answer *"New York"* can be determined.

13.2 Reusing Derivation Results

A proof attempt with a relaxed query differs from the previous attempt only in that the relaxed query clause is a subset of the previous query clause. The FOL representations of the candidate passage and the background knowledge remain unchanged, and thus they have the same logical consequences in both proof attempts. It may therefore seem obvious to reuse the derivation results from the previous proof attempt in the derivation with the relaxed query. E-KRHyper thus offers the functionality to reuse derived consequences in subsequent proof attempts. From an implementational point of view this means preserving the initial hyper tableau branch segment before the first split.

Nevertheless, and perhaps somewhat surprisingly, it turns out that this feature does not offer any clear advantage when processing LogAnswer problems. The reason for this can be found in the characteristics of these problems, which feature a large number of clauses, most of which are irrelevant to a proof. Recall that a proof attempt in LogAnswer starts with an input of about 11,000 clauses. Ideally the query is proven directly by clauses from this initial input, without having to derive any additional clauses. After all, the information retrieval phase of LogAnswer specifically aims at finding candidates that match the query well. In order to find such ideal proofs fast, E-KRHyper always begins the derivation by trying to evaluate the query clause[3] in a hyper extension step on the initial input.

However, when the prover fails to find a proof within the time limit, then it is likely to have derived thousands of new clauses. Indeed, even on the LogAnswer problem test set E-KRHyper on average derives about 8,000 new clauses per problem, despite solving 22.7% of the problems straight from the input. In the relaxation step these derived clauses are added to the original 11,000, effectively becoming new input. Then E-KRHyper begins its derivation by attempting to apply the relaxed query clause to the extended input in a hyper extension step.

[3]Generally the negative clauses are evaluated first, but in the LogAnswer scenario the query clause is the only negative clause.

test type	precision	recall	F-score
information retrieval baseline	0.291	0.098	0.147
shallow feature baseline	0.433	0.421	0.427
no relaxation, logical features	0.702	0.130	0.219
no relaxation, all features	0.449	0.449	0.449
no relaxation, partial results	0.462	0.429	0.445
max. 1 relaxation	0.490	0.390	0.434
max. 2 relaxations	0.565	0.378	0.453
max. 3 relaxations	0.533	0.386	0.447
max. 4 relaxations	0.518	0.402	0.452

Table 13.1: Evaluation results for E-KRHyper in LogAnswer on CLEF 2007 questions

By relaxing the query we have increased its chances of matching the original input, but at the same time we have greatly increased the search space for the computation of the critical first hyper extension step by adding the previous derivation results to the original input. In cases where the relaxed query could have been proven by the original input, the previous derivation results therefore become a distraction that slows down the refutation.

Similar concerns still apply even in the case when a relaxed query actually requires some derived clauses, yet the previous derivation resulted in vastly more than these clauses. All these additions to the input increase the search space, and they can act as a brake when trying to find the proof.

Although the capability to reuse derivation results was built into E-KRHyper specifically for LogAnswer, in the long run we have therefore decided to forgo its usage in LogAnswer. It may still prove advantageous when dealing with large ontologies outside of LogAnswer, where we cannot rely on filtering based on information retrieval. As such it takes part in ongoing experimentation with Cyc and YAGO.

13.3 Evaluation

With relaxation we have reached the aspects of E-KRHyper and LogAnswer that are difficult to evaluate automatically. Relaxation certainly increases the number of proofs the prover finds for a given set of QA problems - as mentioned, without relaxation only 0.7% of the problems occurring during CLEF 2010 would have been solved, while with relaxation this increased tenfold to 7%. However, a proof for a relaxed query may be irrelevant for the original query, and this relevance can only be evaluated outside the prover. Ultimately it is the resulting natural language answer that must be judged, which still necessitates a human assessment.

For this reason we refer to a relaxation-specific evaluation [10] we performed with LogAnswer on questions taken from the CLEF 2007 QA competition. After accounting for unsuitable question types and incomplete parses, in total 12,377 answer candidates were retrieved for 93 questions and subjected to logical processing. According to manual assessment and annotation only 254 of the candidates (2%) actually contain an answer. We measured the precision

(ratio of correct answers found to total answers found), the recall (ratio of correct answers found to correct answers among the candidates) and the F-score (harmonic mean of precision and recall). Several different configurations of LogAnswer were used, and the results are summarized in Table 13.1. Two settings serve as baselines where no logical processing was used: The information retrieval baseline only evaluated candidates according to their *irScore* (see Section 8.4) and the shallow feature baseline considered all shallow features of the candidates. Using IR only yields poor results, but the shallow baseline is remarkably good in comparison. When using the logical processing we distinguish different settings. In the first we accepted only full proofs, and we ranked the proofs according to the logic-based features (again see Section 8.4). This achieved the highest precision, meaning that few false candidates will result in a full proof, but unfortunately the recall was low, indicating that most correct candidates also did not result in a full proof. The next test run was similar to the previous one, but now we also used the shallow features. This lowered the precision less than it increased the recall, so in practice this setting would be preferable, with an F-score more than twice as large as the previous one.

The remaining runs make use of the relaxation mechanism, allowing an increasing number of relaxation iterations per answer candidate. The first of these test runs (*"no relaxation, partial results"*) may need some explanation: While no actual reasoning with a relaxed query was carried out, this run made use of the partial results and could extract *FOCUS* variable bindings from these. The subsequent runs show that allowing up to two relaxations gives the best results overall. While the ideal maximum of relaxations can vary depending on the specifics of a given question set and changing knowledge sources, in general we have kept LogAnswer at using two, sometimes up to three relaxations. Beyond this the results become too unreliable.

Chapter 14

Evaluation of LogAnswer

As mentioned in Section 4.2 it is difficult to evaluate a QA system due to the need to understand an answer in order to assess its accuracy and completeness regarding the question. Technically the developers of a QA system can do this themselves, but this entails issues of possible bias in that answers may be judged too kindly, or even too strictly in an attempt to avoid the former. An external evaluation is therefore preferable. To evaluate our progress with the development of LogAnswer, the system participated in various QA competitions at CLEF, beginning in 2008. The conditions of these competitions vary a lot from year to year, so each will be described in detail, together with the results LogAnswer achieved. However, most of these competitions have some points in common. Each participant receives a collection of knowledge sources, typically corpora in the appropriate language, and a set of usually 200 questions that can be answered with the knowledge from the sources. The participants then have about five days to produce the answers. The QA systems remain and run on their respective home hardware infrastructure. The competition posits no limits on the hardware specifications, so unlike CASC the performance and the results may be influenced by differences in the computer equipment of the participants. Multiple answers per question may be allowed, reflecting the fact that QA systems normally produce several ranked answers for a question. Sometimes questions may be included which cannot be answered with the provided knowledge sources; the systems have to recognize such questions as unanswerable and give an empty answer. The answers are sent to the organizers via email. They are then reviewed by a panel of judges. Participants are usually allowed to submit two independent answer sets (two "runs"), each typically representing a different configuration of their QA system. These runs are evaluated separately.

14.1 The CLEF 2008 Competition - QA@CLEF

2008 marked the first participation of LogAnswer in CLEF, more specifically in the multilingual question answering competition track *QA@CLEF* [FPA+08]. Participants got access to newspaper corpora, and they were allowed to use a static Wikipedia dump in their respective language as well. 200 questions had to be answered, including factoid questions, questions asking for definitions and questions requiring list answers. An answer set for the 200 questions could con-

tain up to three answers for each question. Every question had to be answered with at least one answer, though this could be the empty answer if the system determined the question to be unanswerable with respect to the knowledge sources. Ideally each answer had to be accurate, complete and concise, and it had to be supported by the source passages it was derived from. Deviations from this ideal were penalized in the assessment by the judges.

There were 21 participants in 11 languages, three systems participated in German. Among the 200 German questions, 86 targeted the provided newspaper corpora and 114 related to Wikipedia. Several performance measures were used in the evaluation of each run:

Accuracy: Following the normal understanding of accuracy, this is the ratio of correct answers to the total number of answers. Only the first out of the three answers to each question was considered for this.

Confidence Weighted Score (CWS): This measure takes the n (up to three) different answers for a question into account and is calculated as follows [Voo01]:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{\text{number of correct answers among first } i \text{ answers}}{i}$$

This measure awards systems which rank correct answers highly. The CWS for a run is then the average CWS of its answer tuples for the 200 questions.

Mean Reciprocal Rank (MRR): This is a different measure to account for multiple answers. For each question a score is computed as $1/r$, where r is the rank of the most highly ranked correct answer. If there is no correct answer, the score is zero. The MRR for a run is then the average score for the 200 questions.

Regarding the accuracy, LogAnswer derived 27 correct first answers, while 163 were judged as wrong, and the remaining first answers were not properly supported by sources or they were not sufficiently exact. This resulted in an accuracy of 0.135, i.e. 13.5%. The CWS was 0.029 and the MRR was 0.179. While this performance was unimpressive given the average accuracy of 23,6% in the competition, it was nevertheless promising, considering that the participating LogAnswer prototype had been assembled in a short amount of time. Our evaluation [9] of these results revealed that one weakness of this prototype had had significant impact: The logical evaluation was only used on candidates with perfectly parsed text passages, while incompletely parsed passages were not included in the knowledge base at all. Thus only 60% of the corpora were available to LogAnswer, and any questions that referred to knowledge in the other 40% were impossible to answer. Other weaknesses were found in the ML techniques for the evaluation of shallow and logical features. Also problematic was the answer extraction from certain grammatical constructions like appositions. Recall that while a successful proof binds a MultiNet node identifier to the *FOCUS* variable, the final natural language answer still has to be extracted from the network and the source passage by identifying a suitable word, name or phrase. Thus even a successful proof does not always lead to a correct answer.

14.2 The CLEF 2009 Competition - ResPubliQA

2009 saw significant changes in the organization of the QA competition track of CLEF. The new track *ResPubliQA* [PFS+09] sought to address the variance in the competition conditions between the different languages. While in previous years the organizers had done their best to ensure equal conditions, the fact that each language had its own unique corpora and its own distinct questions made comparisons between QA systems for different languages difficult. For this reason ResPubliQA moved away from newspaper articles and Wikipedia as knowledge sources, and instead the competition was based on a subset of the *JRC-Acquis*[1] corpus, a collection of legislation documents of the European Union (EU). These documents are available in parallel translations in 22 languages of EU member states. 500 questions pertaining these documents were devised, and the participants received the questions and the document collection in their respective languages. The intention was that all participants answer the same questions with the same knowledge, regardless of language.

Answers were no longer required to be as exact as in the previous years. In fact each answer was to be given in the form of a full paragraph from the corpus, not as a compact phrase. Unlike in the previous years only one answer per question was allowed in each run. All questions could be answered from the knowledge in the supplied corpus, hence the empty answer was never the correct choice. However, systems were allowed to abstain from answering questions while still showing their rejected answer candidates. This served to provide insights into the quality of the internal answer evaluation and validation techniques of the systems, and it rewarded systems that remained quiet and abstained from giving uncertain answers over competitors which always answered, even if poorly.

The main evaluation criteria were:

***c@1*:** This score was the primary measure of the competition, calculated as:

$$c@1 = \frac{1}{n}\left(n_c + n_u \frac{n_c}{n}\right)$$

with

n**:** the total number of questions,

n_c**:** the number of questions answered correctly,

n_u**:** the number of unanswered questions.

Accuracy: This measure was modified so that rejected correct answers for unanswered questions were counted as correct answers:

$$\frac{n_c + \text{number of unanswered questions with rejected correct answers}}{n}$$

Different languages may lend themselves to automated processing with varying ease. The competition organizers attempted to lessen the impact of this "language variable" by establishing an IR baseline *c@1* result for all languages, using a standardized information retrieval system. This allowed the computation of a

[1] `http://langtech.jrc.ec.europa.eu/JRC-Acquis.html`

corrected c@1 score for each system, by dividing the *c@1* score by the baseline score for the respective language.

11 QA systems participated in eight languages. LogAnswer was the only German system. We submitted two different runs; each achieved a *c@1* score of 0.44 and an accuracy of 0.4. This was slightly above the competition average of 0.41 for *c@1* and 0.39 for accuracy, and as such it represents a marked improvement in the performance of LogAnswer over the previous year. Going by its corrected *c@1* score of 1.158, LogAnswer was even the third best QA system in the competition. The organizers highlighted LogAnswer's good ability to avoid wrong answers [PFS+09]: In the first run LogAnswer abstained from answering 93 out of the 500 questions, and in 73% of these cases our system was right to do so, as its best answer would have been wrong. In the second run this ratio was even better, as LogAnswer chose not to answer 83 questions, 75% of which would indeed have resulted in a wrong answer otherwise. Only one QA system was better than LogAnswer in this regard with 77% correct rejections. LogAnswer achieved this good rejection rate by using its optional answer acceptance threshold θ (see Section 8.4), adjusted to a value (0.08) determined by machine learning on training data provided by the organizers.

Our evaluation of the results revealed some potential improvements for the future [11]. While the parsing capabilities had been improved over the previous version of LogAnswer, the complex legal language of the sources formed a major obstacle for the parser. Only 26.2% of the competition corpus could be fully parsed, while partial parses covered another 27.8%. Partially parsed passages entered the knowledge base and they enabled some additional shallow validation, an improvement over the earlier LogAnswer which outright discarded passages that were not fully parsed. However, the logical processing remained restricted to fully parsed candidates. Thus the overall employment of logic was actually significantly lower than in the year before. For testing purposes the second submitted run of LogAnswer used no logical processing at all, and it achieved almost as many correct answers as the run with the full system (199 versus 202), which after rounding meant identical *c@1* and accuracy for both runs.

Note of course that this similarity was largely due to the competition requiring only paragraphs as answers, not the exact answers LogAnswer is intended for. While the full LogAnswer run computed the exact answers in the normal way, only the containing paragraphs were then used for the competition results. The logic-less run could not compute exact answers, and had these been a requirement, then this run would not have been an option for the competition. While we were satisfied with the overall performance of LogAnswer, it can be said that the situation cast some doubt on the suitability of CLEF as an evaluation of LogAnswer when used as intended. This view was reinforced by a closer inspection of the questions and the corpus. Often it was clear that the question had been constructed by minor changes to a corpus passage. Compare the following pair of a competition question and a passage from the competition corpus:

Q: *"Which additives may be used in the manufacture of peeled tomatoes?"*

P: *"As additives in the manufacture of peeled tomatoes only citric acid (E 330) and calcium chloride (E 509) may be used."*

Apart from *"which"*, all words from the question also occur in the passage. In fact, the phrases *"may be used"* and *"in the manufacture of peeled tomatoes"* even occur word by word both in Q and P. With data like this the shallow information retrieval can do most of the work by identifying the matching passage. As ResPubliQA required only the paragraph containing the passage instead of an exact answer, the logical processing only provided a nearly superfluous validation of the passage, as there was hardly any reasoning to do. Under such circumstances adding or omitting logical processing cannot make much of a difference, which calls into question the relevance of ResPubliQA for a deduction-based QA system.

14.3 The CLEF 2010 Competition - ResPubliQA

The ResPubliQA competition of CLEF 2010 [PFR+10] remained largely unchanged. The knowledge sources were still based on the JRC-Acquis, now augmented by the *EUROPARL* collection of protocol documents from the European Parliament.[2] These are available in a similar format with parallel translations, but their language style is less formal, so it was hoped this would decrease the difficulties the systems encountered when dealing with the legal texts of the JRC-Acquis in 2009. 200 questions were prepared for the participants, and they were to be answered and evaluated in almost the same way as in 2009, except for the accuracy measure being dropped.

13 QA systems participated in eight languages. There was also an experimental competition track for deriving exact answers instead of paragraphs. However, this track saw only three participants and the results were very poor with an average *c@1* of 0.09, due to the complexity both of the questions and the sources. As our testing showed these difficulties in advance, we had LogAnswer only participate in the regular paragraph answering track, as one of two German language systems in the competition. We submitted two different runs, both using the full LogAnswer system including the logical processing, with one run featuring an improvement in the IR phase regarding the answering of definition questions. LogAnswer achieved a *c@1* score of 0.59 in the normal run and 0.62 in the improved version run. This was better than the German competitor that achieved 0.49 and 0.44 in its runs. The average *c@1* score of the competition was 0.52. Unfortunately the organizers did not compute an independent baseline for all languages, so no language-independent corrected *c@1* score is available. Again LogAnswer showed good performance when it came to avoiding wrong answers, as 80.6% of the 34 rejected answers in the regular LogAnswer run and 79.4% of the 36 rejected answers in the improved version run were indeed incorrect. The θ-threshold had been further optimized by taking into account the results from the previous year. Some systems achieved even higher ratios in this regard, but none of those rejected nearly as many answers, indicating that they only detected and rejected the most obviously wrong answers, and most even had lower *c@1* scores than LogAnswer.

Our evaluation of the results [12] showed that parsing improvements enabled full parses and hence a logical evaluation for 37.0% of the retrieved candidates. The new EUROPARL portion of the corpus did not contribute to this, as only 34.2% of it could be fully parsed, versus 35.1% of the JRC-Acquis portion. Note

[2] `http://www.europarl.europa.eu`

that these numbers are lower than the aforementioned 37.0%, as they refer to the full corpora, not just to the number of retrieved candidate passages. Internal testing revealed that once again the logical processing made little difference. A test run with the logical processing disabled also resulted in a 0.62 c@1 score. The full and the restricted versions of LogAnswer produced different answers in only 14% of the cases. This confirmed our doubts in CLEF as an evaluation venue suitable for a logic-based system, and this was in part responsible for the experiments that will be described in the upcoming Chapter 15.

14.4 The CLEF 2011 Competition - QA4MRE

In 2011 the QA competition track was again subject to major changes. The new competition *QA4MRE* (*Question Answering for Machine Reading Evaluation*) [PHF+11] was built around a task so different from the previous years that it is difficult to regard QA4MRE as a continuation of the earlier QA competitions, or even as a QA competition at all. The participants had to perform a series of tests. In each test a text was supplied, together with 10 questions relating to the text. For each question five possible answers were given by the organizers, only one of them correct, and the task for the QA systems was to identify the correct answer. If a system failed to pick any of the five answers, it was allowed not to answer. Every system had to pass throuch 12 such tests, i.e. it had to analyse 12 texts and 120 questions in total, as well as a total of 600 answer options.

LogAnswer as described in this dissertation was unsuitable for QA4MRE, so a modified version was used instead [13]. The competition made it unnecessary to search for candidate answers. However, the answers provided by the organizers were exact answers, they were not the answer candidates with full text passages that LogAnswer is built to evaluate. This example gives a QA4MRE question and one of the provided answers:

Q: *"What is Bono's attitude with respect to the digital age?"*

A: *"enthusiasm"*

In order to perform a logical evaluation of such answers, LogAnswer combined each answer with the question to form a hypothesis:

H: *"Bono's attitude with respect to the digital age is enthusiasm."*

Each hypothesis then had to be evaluated against the respective test text. The text was transformed into FOL by the usual translation tools in LogAnswer, and then it was attempted to find a proof for the combination of the FOL representations of the hypothesis, the text and the usual background knowledge. Compared to the normal LogAnswer procedure this basically means that the hypothesis had the role of the question and the text served as a candidate passage. A proof showed consistency between text and hypothesis. Relaxation and the evaluation of proofs remained largely as normal, though due to the lack of an IR phase the usage of shallow features was very restricted.

The competition rules allowed the usage of other knowledge sources, but it is unclear how to integrate such sources as the Wikipedia knowledge base of LogAnswer into the hypothesis testing described above. Thus we chose not to

use this option, and instead we augmented the background knowledge by about 20,000 concept subsumption relationships obtained from the free Cyc-fragment *OpenCyc*.[3] This expanded background knowledge is still experimental and not used in the regular LogAnswer.

12 systems participated in three languages, English, German and Romanian. The evaluation measures were *c@1* and accuracy as defined in Section 14.2. In both submitted runs LogAnswer achieved a *c@1* score of 0.22 and an accuracy of 0.18. The average *c@1* in the competition was 0.21. Thus both the general performance and that of LogAnswer were poor, given that random guessing would result in an expected *c@1* score of 0.2. There are several explanations for this. The new competition rules meant that the participants faced a situation that their matured systems were generally not designed for, so for the competition the systems had to be heavily modified, or prototypes were developed. This was aggravated by the lack of training data in the form of results from previous competitions, so methods based on machine learning could not be adequately adapted. Earlier competitions had a focus on factoid questions, whereas the QA4MRE questions were often more complex, and testing the provided correct answers against the texts often required a coherent semantical representation of the full text instead of passages or separate sentences. This was arguably more difficult, and it put more emphasis on coreference resolution than previous competitions. The inadequacy of the systems is therefore in part responsible for the poor overall performance, but some blame must also be placed on the poor quality of the supplied test documents, which contained numerous formatting errors like headlines merged with the subsequent text or even words merged due to missing blanks.

14.5 Conclusions

Apart from the atypical 2011 competition QA4MRE, the participation of LogAnswer at CLEF shows an increase in its performance over the years, with the caveat that a year-by-year comparison is difficult due to differences between the competitions. Nevertheless the results show LogAnswer moving from a below-average performance in its initial 2008 participation to well above the average in 2010. A strength of the system noted by the organizers is its good ability to identify wrong answer candidates and reject them [PFS+09, PFR+10]. Unfortunately this strength could not be used to its potential in 2011 due to the circumstances listed in Section 14.4.

While CLEF remains the main venue for a competitive comparison of QA systems in languages other than English, it is questionable how useful the results are for the evaluation of a logic-based QA system on the web like LogAnswer. The response times in the competitions are much longer than a web usage comparable to a search engine would allow. In the attempt to create equal conditions for systems of all languages, the questions and documents in 2009 and 2010 became so technical in their legal jargon that they are not representative for the NL input a real-world QA system is likely to face. Therefore we began to explore other evaluation methods. The LogAnswer application for the iPhone (see Section 8.2), apart from demonstrating the possible usage of QA on compact mobile devices, was an early attempt to attract external users,

[3] `http://sw.opencyc.org`

so that they would supply us with real-world questions. However, this failed to gain popularity, likely in part due to the admittedly poor performance. We have therefore investigated a different usage scenario and evaluation venue, the internet-based QA forums, and this will be the subject of Chapter 15.

Overall it must be said that while LogAnswer performs well compared to other QA systems as evidenced by CLEF, the performance of QA systems is generally below what would be acceptable for actual usage. The CLEF organizers note that there appears to be an upper bound of 60% accuracy for QA systems [PHF+11], and this is under generous competition time limits. In real-world usage these systems are more likely to give a wrong answer than a correct one, obviously an impediment to widespread adoption.

One aspect that receives little attention in CLEF are the hardware differences, likely because complex QA systems are hardly portable. The competition organizers leave it to the system developers to provide the hardware they deem adequate, which of course in practice is restricted by what the respective groups can actually afford. Thus CLEF has a tendency to downplay or even avoid mentioning hardware, despite the influence of equipment differences potentially outweighing that of other design choices by a sizable margin. This is illustrated by the Watson system (see Chapter 4), which makes extensive use of parallelization on a computer cluster with nearly 3,000 processors. Few QA research groups are likely to have access to this amount of equipment. LogAnswer for example uses hardly any parallelization. Its logical processing is restricted to a single processor, which means that all proof attempts must be carried out in sequence. Watson as used in its victorious *Jeopardy!* participation was capable of answering 85% of questions in at most 5 seconds, yet when using only one CPU it required 2 hours to answer a single question [FBCC+10]. If LogAnswer were to use a comparable computer cluster, then all 200 answer candidates for a question could be processed in parallel. In fact, there would be enough processors to even attempt different axiom selection methods (see Section 12.2) in parallel and to test some relaxed queries in advance. The competition performance of LogAnswer could then be achieved with response times suitable for the web-based use case. With sufficient hardware it could also become unnecessary to reuse input while keeping the parallel provers in continuous operation (see Sections 11.2 and 11.3). Instead the provers could terminate after each proof attempt and then restart to the point where they have indexed the background knowledge. The overarching system would then distribute proof tasks (queries and candidates) to those provers which happen to be done reading the background knowledge.

The Watson system has brought welcome publicity to QA. An unfortunate side effect is that its performance may come to be taken for granted by the public which is unaware of the required hardware infrastructure. Much work remains to be done until QA systems with this level of performance can reach widespread usage.

Chapter 15

LogAnswer and QA Forums

In the previous chapter we mentioned our growing dissatisfaction with CLEF as an evaluation venue for LogAnswer: The test questions are often created by simply transforming an answer-containing sentence from the source documents into question form, so that the question and the passage match in all but a few words. In such cases the shallow retrieval methods do virtually all the work, and the subsequent reasoning is trivial, see Section 14.2. Also, the legal topics favored in CLEF are not representative for a real-world application of a QA system on the web, where it interacts with actual human users.

We have therefore considered combining LogAnswer with QA forums. Such online forums allow their visitors to ask arbitrary questions and to provide answers to each other. Questions are often sorted into browsable categories, and new unanswered questions are listed prominently. A search engine typically allows users to find older questions and answers via keywords. Sometimes multiple answers are allowed for a question, and users can grade and discuss the answers. QA forums exist in many languages, and examples for German forums are *Frag Wikia!*[1], *COSMiQ*[2], *WikiAnswers*[3] and *JustAnswer*.[4] A problem with such forums is that a questioner may have to wait an indefinite time for an answer. Difficult questions may never be answered, and so may even simple factoid questions, for example when other users do not find it worth their time to provide an answer that the questioner could have found with little research in Wikipedia.

A QA system could help here by providing answers almost immediately. Alternatively, since forum visitors do not expect instant answers, the QA system could allow several minutes for a thorough answer derivation and still process every new question being posted on the forum. Even if a QA system may only be able to answer the simplest factoid questions, doing so would nevertheless disburden helpful human participants, allowing them to focus on the more complex and interesting questions. At the same time the development of LogAnswer would profit from a large scale evaluation on questions asked by users who are genuinely interested in the answers. Using automated QA techniques in QA forums is a fairly novel idea. Something similar has been considered before only

[1] `http://frag.wikia.com`
[2] `http://www.cosmiq.de`
[3] `http://de.answers.com`
[4] `http://www.justanswer.de`

by Mihai Surdeanu et al. [SCZ11], but their system does not derive answers like LogAnswer, instead it uses shallow retrieval techniques to identify possibly matching archived answers already phrased by other forum users. While such an approach certainly has its uses, a full QA system that can contribute its own answers without having to rely on the work of human users could become a most valuable contribution to QA forums.

However, we need to pay special attention to potential pitfalls, in particular the problem of wrong answers. In a competition situation like CLEF wrong answers given by a QA system are merely problematic in that they affect the score. In a QA forum with real users on the other hand there is the risk that a QA system will be regarded as a nuisance if a substantial portion of its answers is wrong, as it is common at the current state of the art in QA. As questions are usually marked as answered once they have received their first answer, a QA system could effectively prevent a large number of questions from receiving the attention of other users, despite not helping the original questioner with a correct answer. This kind of QA system integration could devalue a QA forum and drive away its visitors, a situation that should be avoided.

Due to such concerns we have not yet actually connected LogAnswer to a forum at this time. Rather we felt it prudent to perform an extensive evaluation of LogAnswer on the questions already stored in a QA forum, so that we could learn how our system would have to be adjusted to this kind of integration.

15.1 Evaluating Forum Questions

For our first experiments [5] we turned to the German *Frag Wikia!* forum due to its permissive Creative Commons license for its over 100,000 questions. Initially we tested LogAnswer on a random sample of 200 questions taken from this forum.[5] 41% of these contain syntactic mistakes like spelling errors and wrong capitalization. This emphasizes the need for robust NLP components in a QA system, as otherwise the poor orthography will often prevent the parsing of the question, making the strengths of the subsequent processing chain irrelevant in many cases. Fortunately the WOCADI parser of LogAnswer is sufficiently robust to form a FOL representation of 81% of the questions. A manual examination of the sample showed that 61% of the questions have an answer in Wikipedia, despite the policy of *Frag Wikia!* to discourage such questions. For our evaluation this widespread policy violation is positive, as it means that a majority of the *Frag Wikia!* questions can potentially be answered by LogAnswer. When allowing LogAnswer to show up to three answers for each question, then according to our manual assessment of the results our system found at least one correct answer for 30% of the Wikipedia-related questions, or 18% of the full sample.

For a more thorough evaluation [4] we tested LogAnswer on 3,996 hitherto unanswered questions from the *Frag Wikia!* forum. This set contains many difficult and sometimes unclear questions that have accumulated over time without eliciting an answer from human visitors. The idea behind this was to present LogAnswer with the worst-case to be expected in a forum integration, and to judge whether our QA system could be useful immediately by processing a large portion of the backlog of open questions. We used a time limit of 15 seconds

[5] `http://www.loganswer.de/resources/icaart2011.xml`

per question. While this is more than the five seconds applied in the normal web use case of LogAnswer, it is considerably less than the time that could be allowed in an actual forum integration, where even a processing time of several minutes would be acceptable. However, with the large test set this generous limit would have been prohibitively time-consuming, so we chose 15 seconds as a compromise.

The forum questions proved to be difficult to handle. Even after loosening the constraints to count a question as answered correctly if at least one out of four results was correct, LogAnswer achieved an answer rate of only 8.9% for the total test set according to our manual assessment. This corresponds to 21.5% for the questions that can be answered with Wikipedia, which make up about 41% of the test set and thus form a smaller portion among the unanswered questions than in the previous random sample. These numbers may appear disappointing, but one should keep in mind that no human user comes close to answering 8.9% of the questions, much less 18% when considering the forum questions in general as represented by the random sample, not only the unanswered set. Nevertheless the problem remains how to handle the remaining 91.1% or 82% respectively for which LogAnswer could not provide any correct answer. Obviously with these numbers the common QA method of presenting the "best" answer in such cases would be gravely detrimental to the forum experience. While the capability of LogAnswer to derive correct answers may thus already be sufficient for the system to be useful in a QA forum situation, we need to prevent it from posting wrong answers before a forum integration can be considered. Of course the ideal solution would be to improve the answer derivation capability so that more correct answers are posted instead. This is bounded by the general limitations of state-of-the-art QA, though. Apart from that we will see in the following section that there are many forum questions that would be difficult or impossible to answer even for a theoretical ideal QA system. For the time being the more promising approach is to give no answer at all when no correct answer can be derived. We call this *wrong answer avoidance* (WAA).

15.2 Wrong Answer Avoidance

LogAnswer has shown a good capability to reject wrong answers in the CLEF competitions, see Sections 14.2 and 14.3. This was achieved by using the answer acceptance threshold θ (see Section 8.4) with a value computed specifically for the question types and the topics in these competitions, using training data supplied by the organizers. Unfortunately this option was not available when dealing with QA forum questions. While there is a large amount of questions available, the number of correct answers is very low compared to the wrong answers. With only 355 questions receiving at least one correct answer out of four versus 3,641 questions with none, there are not enough positive cases that would enable machine learning to establish a threshold for measuring hundreds of candidates for each question. This would require a clear correlation between the correctness of an answer regarding the question and the shallow and logic-based features of its underlying derivation and proof, something that cannot be ascertained when using our results as training data.

However, although this usual method of establishing a filter does not work as is, we can refine it by adding features specific to the QA forum scenario, pri-

marily features of the questions. This combination of features from the shallow retrieval phase, the logic-based proof and the forum aspects can result in a filter that avoids most wrong answers while minimizing false positives by affecting few of the correct answers. The new question features can be grouped into different classes:

Question Type: The classification of questions into types like factoid questions and definition questions is a standard method in LogAnswer for determining what kind of answer the system should be looking for, see Section 8.4. However, here we can use it to identify questions that are unlikely to be answered correctly, for example questions about opinions. Such questions like *"What is the best compact car"*? are easily recognized by certain keywords or by the use of superlatives, and they clearly go beyond the current scope of QA systems.

Linguistic Problems: There are numerous linguistic problems that can occur in a question, like spelling mistakes, grammatical errors, colloquial language, missing quotation marks or complex questions consisting of multiple sentences. Such problems occur in 50.7% of our test set, and all of them reduce the likelihood of finding a correct answer.

Question Age: The longer a question has remained unanswered in the QA forum, the more likely it is that the question is difficult to answer, and that any answer derived automatically is wrong.

Forum Category: Questioners may place their questions in a category pertaining to the subject matter of the question. Certain categories are notorious for questions that cannot be answered, for example categories about celebrities, where most questioners ask for personal information like phone numbers of pop stars. Another example are software and video game categories, where questions usually are about very specific technical problems that have no answer in Wikipedia. Manually identifying such categories and then simply blocking their questions has been very successful; of 534 questions blocked this way, only 8 were false positives, a rejection precision of 97.8% for this particular feature.

User Features: Some users tend to ask more unanswerable questions than others. Thus a question coming from a user with a high number of unanswered questions is likely to be difficult to answer.

The final WAA filter is a classifier consisting of decision trees that evaluate questions and answers according to these new features and the original shallow and logic-based features. It has been trained on the test set of unanswered questions. Currently it achieves a 90.6% reduction of the false answers, while the number of correct answers is reduced by 58.6%, thus resulting in 3.7% correct answers for the test set of unanswered questions. This corresponds to a precision of 30%, that is, 30% of the answers given are correct, or 147 correct answers are found versus 343 wrong ones. On the one hand this means that the performance is still insufficient for an actual QA forum integration of LogAnswer, as for every correct answer there are more than two wrong ones. On the other hand this is a marked improvement over the unfiltered ratio, where 356 correct answers were offset by 3,640 wrong answers, ten false answers for each correct one. This

shows that we are on the right track. Also, even though giving correct answers to 3.7% of the questions may seem low, it is still better than any single human user, and in a larger QA forum with thousands of new questions every day[6] this nevertheless means a large number of correct answers. Finally one needs to keep in mind that the test set consists of the questions no human user would answer, and many of these are more difficult than the average forum question.

15.3 Towards a Forum Integration

For the time being LogAnswer is not sufficiently reliable to participate in a QA forum without disturbing the users. The WAA filter substantially improves the ratio between correct and wrong answers, but more work is necessary before a forum integration can be considered. The discrepancy between the CLEF competition performance and the performance on QA forum questions highlights the need for real-world evaluations of QA. While it is easy to blame the haphazard writing style and the illusory expectations of many forum visitors for making their questions difficult to handle, a QA system must be prepared to deal with this reality if it is supposed to gain widespread acceptance.

Looking further ahead one can envision another way to use a QA system in a QA forum. As a QA forum is frequented by a multitude of users, the same questions are bound to be asked several times by different people. Most forums try to avoid repeatedly listing such questions as unanswered each time they are posted. Instead an attempt is made to redirect the respective user to an earlier instance of the question, so that old answers can be reused. This requires the forum software to compare the current question with the older questions in the forum archive. Typically this is done by a search based on keywords from the current question. As a result QA forums may not find a semantically equivalent question if it is a paraphrasing of the current question. For example, *Frag Wikia!* contains the question

Q_1: *"What was the name of the first German Chancellor?"*

and asking the same question again will produce this archived question together with an answer. However, rephrasing Q_1 into

Q_2: *"Who was the first Chancellor of Germany?"*

will not lead to the archived Q_1 and its answer, as Q_2 is treated as unique instead.[7] The QA forum *WikiAnswers* shows a similar behaviour: here the paraphrased question is in the archive and will be found when the identical wording is used. The original question is not found, though. Instead the forum search engine suggests other archived questions containing keywords like *"German"* or *"Chancellor"*, but none of them are relevant for the original question.

LogAnswer could offer an improvement by going beyond keywords and instead perform a semantic comparison between questions. The intended result would be similar to that of the aforementioned system by Surdeanu et al. [SCZ11] in that the user could be redirected to existing answers, with the difference that LogAnswer would not rely only on shallow retrieval methods.

[6] *Frag Wikia!*, our forum of choice, only has between 50 and 100 new questions daily.
[7] Note that our rephrased question Q_2 has since been answered by a different user.

This proposed function of LogAnswer would operate as follows. First, a requirement is a new knowledge base derived from the archived questions. For this purpose the questions must be treated like the Wikipedia text passages for the original knowledge base: The questions are parsed and then translated into their MultiNet and FOL representations. The translation largely corresponds to the way LogAnswer would normally translate a question for question answering, with the exception that constants are used instead of variables. Effectively an archived question is treated like a statement in this phase, for example with Q_1 being translated as if it actually read *"Something exists that was the name of the first German Chancellor."*

When a user then asks a new question, LogAnswer can locate potentially matching archived questions using the same information retrieval methods as for the answer candidate passages. E-KRHyper then tests the filtered questions for equivalence to the user question, $Q_1 \leftrightarrow Q_2$. This means that two proofs must be found, one with the user question as a negated conjecture refuted by the archived question and the general background knowledge, and one proof vice versa which refutes the archived question. This is because testing only one direction would not ensure semantic equivalence. For example, while the FOL representation of *"What is a herbivore fish living in the Pacific Ocean?"* could be used to prove *"What is a fish living in the Pacific Ocean?"*, the questions are clearly not equivalent, as the predatory great white shark would be a correct answer to the latter question, but not to the former. Successful proofs in both directions indicate the semantic equivalence of user question and archived question. Relaxation may be used, possibly weakening the probability of the proofs accurately representing equivalence. When several archived questions have been tested this way for one user question, then the ML-based reranking sorts the proof pairs to determine the archived questions with the highest relevance for the questioner. This process is similar to the currently used reranking of proofs and answer candidates, except that only a subset of the criteria can be used since a comparison of questions does not produce an exact answer. Finally LogAnswer presents the best matching questions from the archive, and the user can examine these and any existing answers.

Chapter 16

LogAnswer and Web Services

In its current form LogAnswer is restricted to answering encyclopedic questions, as the answers have to be retrieved from a static knowledge base derived from Wikipedia. This precludes answering questions which do not have permanent answers, like

Q_1: *"What is the weather in Stockholm today?"*

Q_2: *"How much is €2.99 in US-Dollars?"*

Q_3: *"When does the next train to the CeBIT leave?"*

Answering Q_1 requires access to weather data by time and location. For Q_2 a QA system would need current currency exchange rates and arithmetics. Answering Q_3 needs not only time tables, but also the location of the CeBIT trade fair and some method to determine the current location of the user, who probably is only interested in trains that leave from a station nearby. Such information cannot be found in a static knowledge base, and the ability to handle questions of this type would greatly broaden the scope of a QA system, and also likely its appeal. For this reason we have experimented with connecting LogAnswer to web services that can provide such data. According to the W3C,

> *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network."*[1]

While the W3C suggests uniform standards for this interaction, arbitrary services with their own particular operations are acknowledged. For our purposes we therefore use the term for all data resources which provide web-based access in a form that allows automated retrieval. This covers both the *REST*-compliant web services [Fie00] in the strictest sense and other less standardized sources, including web pages that can be accessed and parsed with a reliability that is sufficient to allow obtaining specific data automatically. At the same time we are primarily interested in "factoid" web services that provide their respective data in an almost atomic form which requires little parsing and subsequent processing. For example, while search engines (and even QA systems

[1] `http://www.w3.org/TR/ws-arch`

like LogAnswer itself) could be used as web services, their output is too complex and unstructured for the purposes of this chapter. Instead a web service for LogAnswer should provide data that can be represented in short strings, like *"cloudy"* in response to Q_1, or in numeric form like *"3.74"* for Q_2. The goal is to fill very specific gaps in the knowledge of LogAnswer that become apparent while processing a question, not to supplant the knowledge base or the reasoning in LogAnswer.

A fundamental problem of using web services in QA is their response time. A web service requires an arbitrary amount of time to process a query, and network latency causes additional delays. It is therefore important to avoid unnecessary web service requests, and to use an asynchronous communication with the web services so that the QA system can use the delay phases for other tasks. With the first goal in mind it becomes clear that web services should not be accessed early on in the processing of a question, before its semantics have been analysed. Before this point most web service requests would only amount to pre-emptive guesswork. For example, the question Q_1 could be used to trigger queries containing the word *"Stockholm"* to all sorts of connected web services, even where this makes no sense at all, like a service for measurement unit conversion. Some basic data about the words in the question could reduce obviously futile requests, so that only meaningful requests are sent to services which can actually handle this particular input. However, this still would not avoid requests that have no relevance for the question at hand, like asking a time zone web service for the local time in Stockholm, which is of no interest to someone asking Q_1.

Once the semantics of the full question are known it becomes possible to send requests which could lead to an answer. This may not be enough, though, as questions can still require the knowledge base. For example, to answer Q_3 a QA system could access its own encyclopedic knowledge to determine Hanover as the location and railway station of the CeBIT trade fair, while the current location of the user and the time tables would have to be obtained from suitable web services. The knowledge base is thus still relevant, and the answer candidates may and should have an effect on which requests to send to web services. This means that within the question processing of LogAnswer the web service access should occur only when the semantics of both the question and the candidates are known, for example during the logical processing in E-KRHyper. This way inference conclusions can trigger web service requests, which then may enable new inferences and so on. The result would be a seamless combination of explicit and inferred knowledge from the knowledge base with the knowledge retrieved from web services.

E-KRHyper has been equipped with support for web services. For the time being this support remains on the level of the prover, and the full LogAnswer system does not yet make use of web services. This would require modifications to the knowledge base, which is the work of a knowledge engineer and beyond the scope of this dissertation, and must thus be regarded as future work. In Section 16.1 we describe the formal integration of web services in hyper tableaux. Section 16.2 then explains the implementation of this feature in E-KRHyper. The final Section 16.3 describes an experimental usage of web services for abductive relaxation; this has also been implemented in E-KRHyper.

16.1 Web Services as External Sources in Hyper Tableaux

The integration of web services in E-KRHyper and both the hyper tableaux calculus and the E-hyper tableaux calculus bears similarities to the approach used in the *SPASS-XDB* system [SSW+09, SST+10], which integrates a number of web services and other sources into the SPASS prover and hence the superposition calculus. We adopt the authors' expression of *external sources (of axioms)* to refer to sources such as web services in a more abstract manner. This allows us to use the same techniques both for web services and for any other sources with comparable characteristics, even if they are not web-based, for example a large local database.

We assume that the systems underlying external sources operate on a principle of request and response, i.e. they are accessed by requests, and each request is answered by a response. To formalize this we introduce a new binary predicate symbol $ext/2$ that is used to represent the relation between request and response in the form of an atom $ext(q, a)$, where q is a term representing the request and a is a term representing the associated response. We refer to such atoms as *ext*-atoms, to literals with *ext*-atoms as *ext*-literals, and to unit clauses with *ext*-literals as *ext*-units. An external source like a web service can then be represented as a possibly infinite set of positive *ext*-units which list the requests and the associated responses as provided by the service. There may be multiple different responses $a_1, a_2, \ldots$ for a given request q, represented as multiple *ext*-units:

$$ext(q, a_1) \leftarrow$$

$$ext(q, a_2) \leftarrow$$

$$\vdots$$

Likewise, the same response a may be associated with multiple different requests q_1, q_2 and so on.

The terms q and a can be constructed arbitrarily to properly encode the functionality of the represented source. For example, a meteorological web service suitable for answering the introductory question Q_1 might offer different types of weather data, and it could be represented like this:

$$\vdots$$

$$ext(weather(\text{'Stockholm'}, \text{27-06-2012}), \text{'cloudy'}) \leftarrow$$

$$ext(temperature(\text{'Stockholm'}, \text{27-06-2012}), \text{'15 °C'}) \leftarrow$$

$$\vdots$$

Indeed, it may make sense to represent the identity of the underlying web service as well. This allows representing multiple resources as one external source, for example:

$$\vdots$$

$$ext(weather\_service(weather(\text{'Stockholm'}, \textit{27-06-2012})), \text{'cloudy'}) \leftarrow$$

$$ext(weather\_service(temperature(\text{'Stockholm'}, \textit{27-06-2012})), \text{'15 °C'}) \leftarrow$$

$$ext(currency\_exchange\_service(eur, usd, \textit{2.99}, \textit{27-06-2012}), \text{'\$3.74'}) \leftarrow$$

$$\vdots$$

We assume external sources to consist only of ground positive *ext*-units. This is because the web services we are interested in typically only accept specific requests and thus do not feature a more complex query language that would allow requests with variables. Also, we typically want to avoid variable requests anyway, as we only want to access web services to fill very specific gaps in the knowledge of our QA system. Finally, if it is unavoidable to include a web service which accepts variable requests, then this can be represented by ground positive *ext*-units where fresh constants take the place of the web service variables. For example, the following could be a representation of a variation of the currency exchanger above, where now we allow to leave the target currency unspecified, so that results in different currencies can be returned:

$$\vdots$$

$$ext(currency\_exchange\_service(eur, var_1, \textit{2.99}, \textit{27-06-2012}), \text{'\$3.74'}) \leftarrow$$

$$ext(currency\_exchange\_service(eur, var_1, \textit{2.99}, \textit{27-06-2012}), \text{'£2.39'}) \leftarrow$$

$$\vdots$$

It is then up to the knowledge engineer to design the knowledge base in such manner that variable requests use such constants when accessing an external source.

It must be stressed that the representation of external sources as a set of units is highly idealized, and its purpose is to make such sources accessible within a logical calculus. In practice the full extent of the data that can be provided by a web service is usually not available to a reasoner, as we can only send requests for specific items.

If a web service has no response to some request q, then this is represented by not having any instances of $ext(q, x)$ in the external source. In an implementation such requests will result in some error message by the respective web service, which must then be treated as a non-match by the interface between web service and prover.[2]

[2] In certain circumstances though one might opt to represent some error responses as special FOL terms, as it sometimes might be useful to distinguish between requests that are refused by a web service (for example due to syntax errors) and requests that cannot be answered (for example because the web service is missing data about the particular requested item).

We assume further our extended hyper tableaux calculi to remain time-agnostic. All *ext*-units of external sources are always true. Where this is a problem, a time stamp encoding as in the examples above may help. The QA system can then use its internal time or some other temporal data to time stamp its requests, thereby ensuring that only valid responses are returned. Of course time remains a problematic issue, as external sources may gain new *ext*-units and lose old ones over time. For example, the currency exchange service will not contain the exchange rates of tomorrow before tomorrow, and it might not keep an archive of past exchange rates. This means that the same request may sometimes result in a response and sometimes not. For the formalization we therefore assume an external source to be constant. If in reality it changes over time, then from the formal point of view its changed representation must be regarded as a different external source. In practice it is also possible that a web service changes its inventory of replies during the time of a proof attempt. While this is unlikely with the short time slices given to reasoning in LogAnswer, it cannot be ruled out entirely. We do not account for this in our formalization, as any attempt to deal with this remote possibility on the formal level is likely to overcomplicate the calculus extension beyond its benefits.

An external source can be accessed during the derivation by instantiating the atom of a negative *ext*-literal in a selected clause by a positive *ext*-unit from the external source, provided that both atoms in question have the same ground request term q. The requirement of ground requests is due to web services normally not allowing underspecified requests with variables. We also want to avoid grounding a variable request term with arbitrary terms from the Herbrand universe in a manner similar to purification, see Sections 6.1.2 and 6.2.1. Otherwise this could trigger an excessive amount of time-consuming web service accesses, and it would go against our intention to fill only specific gaps in the knowledge base. The response term a' in the negative *ext*-literal may be a variable, a ground term, or a complex term containing an arbitrary number of variables. As the response term a in the external source must instantiate a', careful construction of the response terms can be used to restrict which responses are acceptable for a given request.

The following example illustrates the principle with a simplified set of clauses pertaining to the introductory question Q_3:

$C_1^{\boldsymbol{ext}}$**:** *ext*(*user_location_service*, *'Cologne'*) ←

$C_2^{\boldsymbol{ext}}$**:** *ext*(*next_train_finder_service*(*'Cologne'*, *'Hanover'*), *'15:05'*) ←

— — — — — — — — — — — — — — — — — — — —

C_1**:** *at*(*'CeBIT'*, *'Hanover'*) ←

C_2**:** *next_train_to*(*Event*, *Time*) ←
at(*Event*, *ToCity*),
ext(*user_location_service*, *FromCity*),
ext(*next_train_finder_service*(*FromCity*, *ToCity*), *Time*)

Clauses above the dashed line represent the external source, clauses below are the clauses fully available to the calculus and the prover. To determine the time of the next train departure to the CeBIT by clause C_2, the knowledge base fact C_1 is used to find the city of Hanover as the target location, thus instantiating the variable *ToCity* with *'Hanover'*. The current location of the user, in this case the city of Cologne, is obtained by accessing a web service which requires no specific input parameters, represented by C_1^{ext}, and instantiating *FromCity* with *'Cologne'*. With both the current location and the destination known the *next_train_finder_service* can now be accessed to retrieve the departure time. This is represented by C_2^{ext}, and it instantiates the *Time* variable with *'15:05'*. Finally the new fact *next_train_to*(*'CeBIT'*, *'15:05'*) is derived, which can be used to produce the answer to Q_3.

In the following we will extend both the hyper tableaux calculus and the E-hyper tableaux calculus with inference rules allowing such derivations. We show the soundness of both extensions. Unfortunately neither is complete, and we will argue why true completeness is unattainable in any approach that seeks to integrate external sources into reasoning, though we offer some approaches to "approximate completeness".

The extension for E-hyper tableaux is described first, as the design of this calculus with its multitude of inference rules lends itself to an extension by a single compact rule. In the generally simpler equality-free hyper tableaux calculus we aim to stay with a single inference rule overall by describing a modified hyper extension step. By necessity this more powerful rule will be more complex to describe, as it effectively encompasses the entire calculus.

16.1.1 External Sources in E-Hyper Tableaux

The ext-sup-left rule (*external superposition left*) selects a negative *ext*-literal to send a request to an external source, and then it applies the response as a substitution to the selected clause. Recall that we use the equational notation with the special constant $\mathbf{t}$ to write originally non-equational atoms as equations.

$$\text{ext-sup-left}(\sigma)\quad \frac{\mathcal{A} \leftarrow \mathit{ext}(q, a') \simeq \mathbf{t}, \mathcal{B} \qquad \mathit{ext}(q, a) \simeq \mathbf{t} \leftarrow}{(\mathcal{A} \leftarrow \mathcal{B})\sigma}$$
$$\text{if} \begin{cases} \mathit{ext}(q, a) \text{ is ground, and} \\ \sigma \text{ is a substitution with } a'\sigma = a. \end{cases}$$

If the ext-sup-left rule is applied with clause C as left premise, D as right premise, the substitution σ and the conclusion E, then this inference instance will be denoted by $C, D \Rightarrow_{\text{ext-sup-left}(\sigma)} E$.

The new inference rule will be encapsulated in a corresponding tableau extension rule. Let $\mathcal{C}$ be a set of clauses, and let $\mathcal{C}^{ext}$ be an external source, a set of positive ground *ext*-units. If $\mathbf{T}$ is an E-hyper tableau for $\mathcal{C}$ with branch $\mathbf{B}$, then $\mathbf{T}$ can be extended by application of the following Ext-Access-extension rule (*external access*):

$$\text{Ext-Access}\quad \frac{\mathbf{B}}{\mathbf{B} \cdot E}\quad \text{if} \begin{cases} \text{there is a clause } C \in \mathbf{B}, \\ \text{a positive ground } \mathit{ext}\text{-unit } D \in \mathcal{C}^{ext}, \text{ and} \\ \text{a substitution } \sigma \text{ such that} \\ \quad C, D \Rightarrow_{\text{ext-sup-left}(\sigma)} E, \text{ and} \\ \mathbf{B} \text{ contains no variant of } E. \end{cases}$$

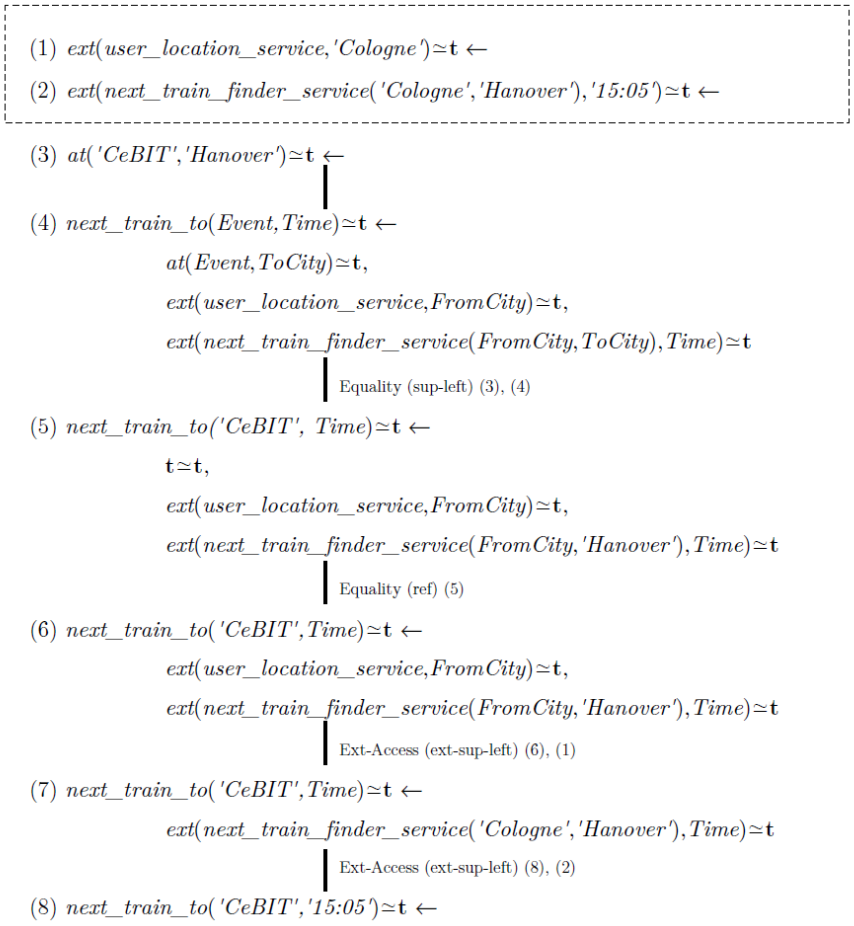


Figure 16.1: Example E-hyper tableaux derivation, dashed box contains external source

Figure 16.1 shows an example of the usage of the new rules. The clauses (1) to (4) are the equational representations of clauses pertaining to the introductory question Q_3, with the dashed box containing the external source. The departure time of the next train to the CeBIT fair is derived in four steps. First a conventional sup-left application with the clauses (3) and (4) instantiates the variable *Event* with *'CeBIT'* while leaving the trivial equational atom $\mathbf{t} \simeq \mathbf{t}$ in clause (5). This literal is dropped in an application of ref, resulting in clause (6). Clause (7) is then derived by accessing the external source to find the current location of the user, in this case *'Cologne'*. With this the request term in the final *ext*-literal is ground, so another query to the external source retrieves the time *'15:05'*, thus resulting in clause (8).

We now prove the soundness of the extended calculus. Let us first clarify the relationship between a set of clauses $\mathcal{C}$ and an external source $\mathcal{C}^{ext}$ with respect to E-hyper tableaux. As $\mathcal{C}^{ext}$ is a set of positive ground *ext*-units, $\mathcal{C}^{ext}$ is satisfiable and so are all its elements. Technically we could therefore regard $\mathcal{C}^{ext}$ as any

other input clauses, and an E-hyper tableaux derivation for $\mathcal{C}$ that accesses $\mathcal{C}^{ext}$ could be seen as an E-hyper tableaux derivation for $\mathcal{C} \cup \mathcal{C}^{ext}$. However, recall that the initial E-hyper tableau of any derivation consists of a single branch of all input clauses. As $\mathcal{C}^{ext}$ may be infinite, this leads to a problem in that even refutational tableaux could contain infinite branches, which is difficult to reconcile with the basic E-hyper tableaux calculus. We therefore keep $\mathcal{C}$ and $\mathcal{C}^{ext}$ separate even on the formal level. An E-hyper tableaux derivation for a clause set $\mathcal{C}$ with an external source $\mathcal{C}^{ext}$ initializes the tableau only with $\mathcal{C}$, and if a branch $\mathbf{B}$ is unsatisfiable, then this may be because $\mathcal{C}$ itself is internally inconsistent (i.e. it is unsatisfiable without accessing $\mathcal{C}^{ext}$), or because $\mathbf{B} \cup \mathcal{C}^{ext}$ is unsatisfiable. This can be written as $\mathbf{B}$ or $\mathcal{C}$ being *unsatisfiable with respect to* $\mathcal{C}^{ext}$, although when it is clear from the context that $\mathcal{C}^{ext}$ may be involved, then we will merely call $\mathbf{B}$ or $\mathcal{C}$ unsatisfiable.

Theorem 16.1 (Soundness of E-Hyper Tableaux with External Sources)**.** *Let $\mathcal{C}$ be a finite clause set and let $\mathcal{C}^{ext}$ be an external source. If the E-hyper tableaux calculus extended by the* Ext-Access*-rule derives a refutation for $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$, then $\mathcal{C}$ is unsatisfiable with respect to $\mathcal{C}^{ext}$.*

Proof. We first show that the new rules ext-sup-left and Ext-Access preserve satisfiability. Let $C, D \Rightarrow_{\text{ext-sup-left}(\sigma)} E$ with $C = \mathcal{A} \leftarrow \mathit{ext}(q, a') \simeq \mathbf{t}, \mathcal{B}$, the *ext*-unit $D = \mathit{ext}(q, a) \simeq \mathbf{t} \leftarrow$, and the conclusion $E = \mathcal{A} \leftarrow \mathcal{B}$. Assume the premises C and D to be satisfiable with a model I. From $a'\sigma = a$ it follows that $(\mathit{ext}(q, a') \simeq \mathbf{t})\sigma = D$. As $I \models D$ this means that I is not a model for the selected negative *ext*-literal of C, i.e. $I \not\models \neg \mathit{ext}(q, a') \simeq \mathbf{t}$. Instead I must be a model for some of the other literals of C, meaning $I \models \mathcal{A} \leftarrow \mathcal{B}$. Therefore it also holds that $I \models (\mathcal{A} \leftarrow \mathcal{B})\sigma$, and hence I is also a model for the conclusion E. As an immediate consequence it follows that the extension rule Ext-Access also preserves satisfiability. The contrapositive of this is that if the conclusions of the new rules are unsatisfiable, then so are the premises.

Now, let $\mathbf{T}$ be the closed tableau of the refutation of $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$. From the contrapositive above and the soundness of the basic E-hyper tableaux calculus we conclude that if a tableau $\mathbf{T}_i$ of a derivation contains only branches that are unsatisfiable , then so does the predecessor $\mathbf{T}_{i-1}$. The closed $\mathbf{T}$ contains only unsatisfiable branches, and by induction on the length of the refutation we conclude that the initial tableau $\mathbf{T}_0$ which consists of one branch with the clauses from $\mathcal{C}$ is unsatisfiable with respect to $\mathcal{C}^{ext}$. □

It may be remarked that ext-sup-left is essentially a special case of sup-left, restricted to work only with *ext*-literals that meet the groundness conditions regarding the request and response terms. This is easy to show: Let $C, D \Rightarrow_{\text{ext-sup-left}(\sigma)} E$ as above, and let us disregard the sources of these clauses, for which the purely clause-based inference rules specify no requirements anyway. Then it is also possible to carry out $C, D \Rightarrow_{\text{sup-left}(\sigma)} E'$ with $E' = (\mathcal{A} \leftarrow \mathbf{t} \simeq \mathbf{t}, \mathcal{B})\sigma$, because all conditions of sup-left are met:

1. $\mathit{ext}(q, a')$ is not a variable,
2. the substitution σ is also a most general unifier of $\mathit{ext}(q, a)$ and $\mathit{ext}(q, a')$,
3. $\mathit{ext}(q, a)\sigma \not\preceq \mathbf{t}\sigma$ as $\mathbf{t}$ is the smallest term in the term ordering,
4. $\mathit{ext}(q, a')\sigma \not\preceq \mathbf{t}\sigma$ as $\mathbf{t}$ is the smallest term in the term ordering.

The equation $\mathbf{t} \simeq \mathbf{t}$ in E' is trivial and can be dropped by reflexivity or simplification, ultimately resulting in the same clause E as the conclusion of ext-sup-left above. Thus, if the *ext*-units from the external source were part of the normal input, then the functionality of the new rules would be subsumed by the basic E-hyper tableaux calculus.

On the converse this means that the external source can basically be regarded as a special subset of the total input clauses, a subset that can only participate as premises in special cases of sup-left, which is an indicator as to why this solution is not complete. We will first extend the original hyper tableaux calculus in a similar manner and then discuss the issue of incompleteness for both extensions.

16.1.2 External Sources in Hyper Tableaux

In order to deal with external sources in the hyper tableaux calculus without equality we modify its only inference rule, the hyper extension step. Since this rule can be seen, loosely speaking, as a combination of multiple sup-left and ref applications followed by split (see Section 6.2.9), the modified hyper extension can be regarded in a similar manner as now also incorporating the new rules ext-sup-left and Ext-Access. Basically the new *hyper extension step with external access* provides the option for the atoms of negative *ext*-literals to unify not only with branch literals, but also with *ext*-units from an external source, provided the request terms are ground. As hyper extension computes one simultaneous most general substitution for all negative literals of the extending clause, originally non-ground request terms may become ground during this computation. This adds some formal complication to the definition of this inference rule, even though its operation is likely quite obvious.

The modified hyper extension works mostly like the original, except that *ext*-atoms from body literals in the extending clause now have the option to unify with units from the external source, provided the request terms in these atoms are ground before their respective unification. This may be because they were already ground in the first place in the extending clause, or because their variables got instantiated during the substitution computation. A variable may have become instantiated either because it also occurs in some other literal of the extending clause, where it got instantiated by unification with a branch literal, or because it also occurs in the response term of some other *ext*-literal which instantiated them by accessing the external source with a ground request term.

Let $\mathcal{C}$ be a finite clause set and let $\mathcal{C}^{ext}$ be an external source of ground *ext*-units. Hyper tableaux for $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$ are inductively defined as follows:

Initialization step: A one node literal tree is a hyper tableau for $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$. Its single branch is labeled as open.

Hyper extension step with external access: This inference requires the following conditions:

1. $\mathbf{B}$ is an open branch with the leaf node $\mathbf{N}$ in the hyper tableau $\mathbf{T}$.
2. $C = \mathcal{A} \leftarrow \mathcal{B}$ is a clause from $\mathcal{C}$ (referred to as the *extending clause*) with $\mathcal{A} = \{A_1, \ldots, A_m\}$ $(m \geq 0)$ and $\mathcal{B} = \{B_1, \ldots, B_n\}$ $(n \geq 0)$.

3. σ is a most general substitution such that $[\![\mathbf{B}]\!] \cup \mathcal{C}^{ext} \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$; in particular, with every $B_i \in \mathcal{B}$ associate the specific literal or unit clause L_i that forms this model, i.e. $L_i \models B_i\sigma$ and $L_i \in [\![\mathbf{B}]\!] \cup \mathcal{C}^{ext}$.
4. Let V be the *set of branch-instantiated variables*, which is defined as:
   (a) $x \in V$ if x occurs in some $B_i \in \mathcal{B}$ with $L_i \in [\![\mathbf{B}]\!]$ such that there is a most general substitution γ with $L_i \models B_i\gamma$ and $x\gamma$ is ground and there is a possibly empty substitution δ such that $\gamma\delta = \sigma$ (x is directly branch-instantiated).
   (b) If there is a $B_i = ext(q_i, a_i) \in \mathcal{B}$ with $L_i \in \mathcal{C}^{ext}$ and for every x occurring in q_i it holds that $x \in V$, then for every y occurring in a_i it holds that $y \in V$ (y is indirectly branch-instantiated).

   Then for every $B_i = ext(q_i, a_i) \in \mathcal{B}$ with $L_i \in \mathcal{C}^{ext}$ and for every variable x occurring in q_i it must hold that $x \in V$.
5. π is a purifying substitution for $C\sigma$.

If all the above conditions hold, then the literal tree $\mathbf{T}'$ is a hyper tableau for $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$, where $\mathbf{T}'$ is obtained from $\mathbf{T}$ by attaching $m + n$ child nodes $M_1, \ldots, M_m, N_1, \ldots, N_n$ to $\mathbf{B}$ with respective labels $A_1\sigma\pi, \ldots, A_m\sigma\pi, B_1\sigma\pi, \ldots, B_n\sigma\pi$ and labeling every new branch $(\mathbf{B} \cdot M_1), \ldots, (\mathbf{B} \cdot M_m)$ with positive leaf as open and every new branch $(\mathbf{B} \cdot N_1), \ldots, (\mathbf{B} \cdot N_n)$ with negative leaf as closed.

As an example, recall this introductory clause set for question Q_3:

C_1^{ext}: *ext*(*user_location_service*, *'Cologne'*) ←

C_2^{ext}: *ext*(*next_train_finder_service*(*'Cologne'*, *'Hanover'*), *'15:05'*) ←

– – – – – – – – – – – – – – – – – – – –

C_1: *at*(*'CeBIT'*, *'Hanover'*) ←

C_2: *next_train_to*(*Event*, *Time*) ←
at(*Event*, *ToCity*),
ext(*user_location_service*, *FromCity*),
ext(*next_train_finder_service*(*FromCity*, *ToCity*), *Time*)

The derivation is shown in Figure 16.2. The unit C_1 has been added as a literal to the branch right away. Then C_2 is selected as an extending clause. The atom of its first negative literal unifies with the branch literal *at*(*'CeBIT'*, *'Hanover'*). This instantiates the variable *Event* with *'CeBIT'* as well as *ToCity* with *'Hanover'*.

The second negative literal of C_2, $\neg$*ext*(*user_location_service*, *FromCity*), already has a ground request term, and this is used to retrieve *'Cologne'* from the external source, instantiating the variable *FromCity*.

Thus $\neg$*ext*(*next_train_finder_service*(*FromCity*, *ToCity*), *Time*), the third negative literal of C_2, can now be regarded as having a ground request term. This

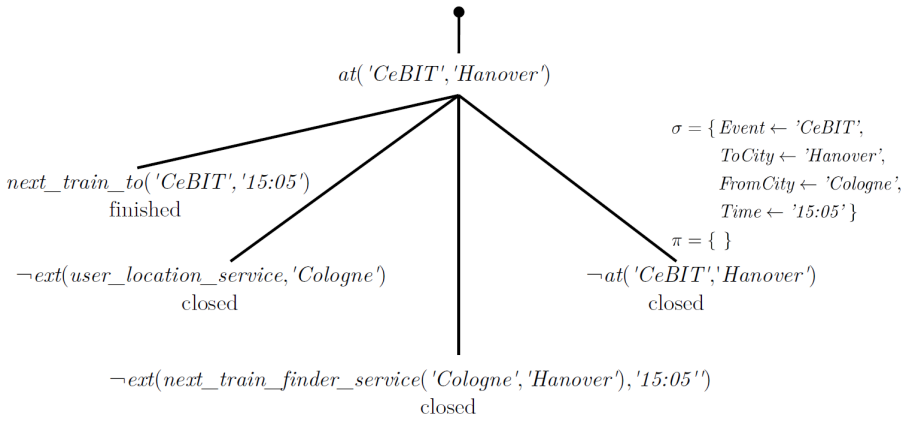


Figure 16.2: Example hyper tableaux derivation

enables another request to the external source which instantiates the variable *Time* with *'15:05'*. All negative literals of C_2 have now been refuted using the common unifier σ. No purification is required, so π remains empty, and every σ-substituted literal is added as a new leaf, with the negative leaves closing their branches immediately. Only the branch for *next_train_to*(*'CeBIT'*, *'15:05'*) remains open. In the presence of a query clause this branch could likely be used for an answer.

Note that the modified hyper extension with external access does not require negative *ext*-literals to be refuted by the external source. Rather, if the branch contains positive *ext*-literals, then it is possible for them to refute literals of the extending clause just as non-*ext*-literals do. Similar to the extension of E-hyper tableaux the modified hyper extension is therefore subsumed by the original when the clauses from the external source are treated as ordinary input clauses.

We now prove the soundness of hyper tableaux with external sources.

Theorem 16.2 (Soundness of Hyper Tableaux with External Sources)**.** *Let $\mathcal{C}$ be a finite clause set and let $\mathcal{C}^{ext}$ be an external source. If the modified hyper tableaux calculus extended by the hyper extension step with external access derives a refutation for $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$, then $\mathcal{C}$ is unsatisfiable with respect to $\mathcal{C}^{ext}$.*

Proof. We first show that the hyper extension step with external access preserves satisfiability. Let $\mathbf{B}$ be an open branch in a hyper tableau T for a finite clause set $\mathcal{C}$ and an external source $\mathcal{C}^{ext}$ of positive ground *ext*-units. Let $C = \mathcal{A} \leftarrow \mathcal{B}$ with $\mathcal{A} = \{A_1, \ldots, A_m\}\,(m \geq 0)$ and $\mathcal{B} = \{B_1, \ldots, B_n\}\,(n \geq 0)$ be a clause from $\mathcal{C}$ that serves as an extending clause in a hyper extension step with external access, using a most general unifier σ and a purifying substitution π. Assume C to be satisfiable with a model I. $[\![\mathbf{B}]\!] \cup \mathcal{C}^{ext}$ consists only of positive unit clauses and is therefore satisfiable. Since $[\![\mathbf{B}]\!] \cup \mathcal{C}^{ext} \models \forall(B_1 \wedge \ldots \wedge B_n)\sigma$, on the converse it must hold that $\neg B_1\sigma \vee \ldots \neg B_n\sigma$ is unsatisfiable. Thus $I \not\models \neg B_1 \vee \ldots \neg B_n$, and instead it must hold that $I \models A_i$ for some $A_i \in \mathcal{A}$ for I to satisfy C. Then it also holds that $I \models A_i\sigma\pi$, and the new branch $\mathbf{B} \cdot M_i$ resulting from extending $\mathbf{B}$ by the node M_i labeled with $A_i\sigma\pi$ is satisfiable.

The contrapositive of the above is that if a hyper extension step with external access extends a branch $\mathbf{B}$ with no satisfiable branches, then $\mathbf{B}$ and the extending clause, the premises of this extension, are unsatisfiable with respect to $\mathcal{C}^{ext}$, too. Let therefore $\mathbf{T}$ be the closed tableau of the refutation of $\mathcal{C}$ with respect to $\mathcal{C}^{ext}$. From the contrapositive above we conclude that if a tableau $\mathbf{T}_i$ of a derivation contains only branches that are unsatisfiable , then so does the predecessor $\mathbf{T}_{i-1}$. The closed $\mathbf{T}$ contains only unsatisfiable branches. By induction on the length of the refutation we conclude that the premises of the first hyper extension step, i.e. the first extending clause and the empty initial branch, are unsatisfiable with respect to $\mathcal{C}^{ext}$. Hence $\mathcal{C}$ is unsatisfiable, too. □

As with the extended E-hyper tableaux this solution is not complete, an issue that will be discussed in the next section.

16.1.3 Incompleteness

Neither of the calculus extensions is complete, and in general they cannot be used for model generation, as they may end up with finite branches that are not closed, despite the input clauses being unsatisfiable with respect to the external source. We identify the reasons for this, arguing that completeness is unattainable for external sources in general, regardless of the calculus. We also discuss some workarounds and the trade-offs they involve.

There are two interrelated, fundamental obstacles to completeness:

***unrequestable responses*:** External sources like web services typically only respond to specific requests, which the calculus extensions reflect by the need to ground request terms before accessing the external source. This means that only those responses can enter the reasoning for which the exact request can be formed beforehand. All others remain external, even though they might contribute to a proof.

***variable request terms*:** It is not always known before the reasoning what requests should be sent to external sources, and for efficiency reasons we want to avoid sending irrelevant requests, for example by exhaustively forming requests beforehand, outside the reasoner. Thus in order to use external sources in a flexible way, it must be possible to form requests dynamically during the reasoning process, and this requires variables in the clauses which can then be instantiated. However, this results in a contradictory situation: Logically variables are more "powerful" than ground terms - they can represent whole sets of terms, and they can subsume terms. Yet when dealing with external sources a ground term is more useful than a variable, because a ground term may be used as a request, while a variable may not.

Neither of these can be circumvented due to the reality of web services. The technical limitations to their accessibility prevent approaches for theory integration [Fur94] which require sources with faster and richer interfacing, making them more suitable for local deductive databases. In comparison web services only offer a keyhole access to their data, and any formal integration which ignores this restriction is destined to be inapplicable in practice.

We will use a series of examples to illustrate the problems involving completeness.[3] The first shows in a most compact form how the technical limitations can prevent a refutation:

Example 16.1.

C_1^{ext}**:** $ext(q, a) \leftarrow$

– –

C_1**:** $\leftarrow ext(x, y)$

As before, the dashed line separates the external source above from the ordinary input clauses below. q and a are ground terms as is customary for external sources. Obviously, without this separation the clauses C_1 and C_1^{ext} together would be unsatisfiable, but since C_1 has no ground request term, C_1^{ext} is inaccessible. Contrast this with:

Example 16.2.

C_1^{ext}**:** $ext(q, a) \leftarrow$

– –

C_2**:** $\leftarrow ext(q, y)$

Here C_2 can access the external source and a refutation can be derived, which makes the first failure all the more aggravating, given that C_2 is an instance of the more general C_1. Worse, even the following example cannot be refuted:

Example 16.3.

C_1^{ext}**:** $ext(q, a) \leftarrow$

– –

C_1**:** $\leftarrow ext(x, y)$

C_3**:** $p(y) \leftarrow ext(q, y)$

Here only C_3 can access the external source, using it to derive $p(a)$, while the refutation with C_1 still fails due to its variable request term. This example shows that we cannot simply disregard some web service data as unreachable and thereby irrelevant, because clearly a is retrieved, just not wherever it is needed.

[3] These examples are numbered individually due to their frequent cross-referencing, unlike previous examples which were relevant mostly for their immediate context.

Restricting Completeness

The authors of the aforementioned SPASS-XDB acknowledge the problem of completeness [SSW+09, SST+10]. In their system the responses from external sources are asserted as unit axioms, effectively treating them as ordinary input clauses. Under these conditions the authors offer a notion of completeness with respect to the delivered responses, in that their system performs the same deductions as if the delivered responses were ordinary input axioms right from the start [SSW+09]. We will refer to this kind of completeness as *response-completeness*, and we can achieve it in hyper tableaux by a minor addition to the external access rules in both calculus extensions: Whenever an *ext*-unit is accessed in the external source, then it is also added as a positive literal (unit clause) to all branches in the hyper tableau (E-hyper tableau). This can be done either by appending it to all open branches, or by prepending it to the entire tableau, essentially having it form a new root node. This way the *ext*-unit becomes indistinguishable from an input clause, and as both hyper tableaux calculi are complete for input clauses, their response-completeness trivially follows.

However, there are drawbacks to this. By permitting an inference in one branch to affect all branches, the hyper tableaux calculi no longer allow processing one branch at a time. Consider the following example:[4]

Example 16.4.

$C_1^{\boldsymbol{ext}}$**:** $ext(q, a) \leftarrow$

C_4**:** $\leftarrow ext(x, y), p(x)$

C_5**:** $p(x), p(q) \leftarrow$

Here C_5 is used to split the tableau. Let us assume the first branch $\mathbf{B}_1$ to use $p(x)$ as the split literal; this allows no ground instantiation of the request term in C_4, and thus C_1^{ext} cannot be used to close this branch immediately. The other split branch $\mathbf{B}_2$ uses $p(q)$, by which the request term in C_4 can be instantiated, eventually resulting in a closing of that branch. By making C_1^{ext} available to all branches as ordinary input, the $\mathbf{B}_1$ can now be closed as well. In an implementation $\mathbf{B}_1$ would have to be postponed once its computation reaches a point where it is not closed, yet its derivation cannot continue. It could only be revisited once other branches have added potentially useful *ext*-units to all branches. Clearly this is undesirable, as storing unfinished branches for later computation can use large amounts of memory.

[4] Most provers would likely reduce clause C_5 to the unit $p(x) \leftarrow$, but we disregard such optimizations here to keep the example both simple and functional regarding the demonstration of the problem. More elaborate examples could be constructed to exhibit the same problem under optimizitations.

Also, there is of course no guarantee that a later branch will add a useful *ext*-unit, as shown in this modified example:

Example 16.5.

$C_1^{\boldsymbol{ext}}$**:** $ext(q, a) \leftarrow$

— — — — — — — — — — — — — — — — — — — —

C_4**:** $\leftarrow ext(x, y), p(x)$

C_6**:** $p(x), p(y) \leftarrow$

Here both split branches end up in the same situation with a non-ground request term. As the tableau has then reached a fixed-point, one might consider both branches to represent models under the notion of response-completeness, regarding C_1^{ext} as irrelevant due to being unreachable. However, the computation of a tableau is not guaranteed to reach a fixed-point where no branches can be extended, because it is possible to have branches that can be extended to infinity. This means that branches with variable requests may be postponed indefinitely, which limits the usefulness of response-completeness for model generation purposes. Also, response-completeness does not resolve the discrepancy between the first two Examples 16.1 and 16.2, as the former cannot be refuted while the latter can, despite the latter being an instance of the former.

Forming Requests by Domain Enumeration

An alternative method to lessen the problem of unrequestable responses is to apply a *modified range-restriction transformation* to the input clause set $\mathcal{C}$. This is done by the following steps:

1. Add the domain clauses to enumerate the Herbrand-domain of $\mathcal{C}$ using the special *dom*-predicate (see Section 7.4.2).

2. For every clause $C \in \mathcal{C}$ with $C = \mathcal{A} \leftarrow \mathcal{B}$, replace C with $C^{rr} = \mathcal{A} \leftarrow \mathcal{B}, dom(x_1), \ldots, dom(x_n)$ $(n \geq 0)$, where for each x_i $(0 \leq i \leq n)$ one of the following holds:

   - $x_i \in vars(\mathcal{A})$ and $x_i \notin vars(\mathcal{B})$, or
   - $x_i \in vars(q)$ for some $ext(q, a) \in \mathcal{B}$.

We apply this transformation to the clauses from Example 16.3, resulting in this new example:

Example 16.6.

$C_1^{\boldsymbol{ext}}$**:** $ext(q, a) \leftarrow$

$C_1^{\boldsymbol{rr}}$**:** $\leftarrow ext(x, y), dom(x)$

$C_3^{\boldsymbol{rr}}$**:** $p(y) \leftarrow ext(q, y)$

$C_1^{\boldsymbol{dom}}$**:** $dom(q)$.

With C_1^{dom} it is now possible to ground-instantiate the request-term x in C_1^{rr} to q, thereby allowing this clause to access C_1^{ext} and to derive a refutation. This transformation can also help in cases like Example 16.4, where the domain enumeration makes symbols and terms from all branches available in every single branch. This limits the need to postpone branches, because requests can be formed from symbols normally occurring only in later branches.

There are two disadvantages to this approach. First, the Herbrand universe may be very large, and when there are only few requests that can actually result in responses, then a lot of time may be wasted processing futile requests from the exhaustive domain enumeration. This is undesirable given the latency of web service transactions, and it runs against our intention to only fill specific gaps in the knowledge rather than guessing requests.

Second, the domain enumeration is restricted to symbols from the input clauses, as the symbols from the external source are not known, and even if they were, there might be an infinite number of them. This issue can be alleviated to some degree by adding domain clauses for symbols that are returned in responses from the external sources. This allows using symbols from the external source to form new requests. However, the request terms and the response terms in an external source may draw their symbols from separate sets, and it is not certain that access to response symbols will eventually allow to form all valid requests. Example 16.1 thus still cannot be refuted.

Problems of Redundancy

Two problems deserve further mentioning, as they can prevent successful retrieval from external sources even though all required clauses and terms are in place. Both are caused by optimizations which serve to curtail the number of clauses by detecting and eliminating those which have become redundant. The first can occur due to rewriting, because the *ext*-units in the external source cannot serve as targets for such operations like the unit-sup-right inference rule.

The following example illustrates this problem:

Example 16.7.

C_1^{ext}: $ext(f(q), a) \leftarrow$

– – – – – – – – – – – – – – – – – – – –

C_1: $\leftarrow ext(x, y), p(x)$

C_2: $p(f(q)) \leftarrow$

C_3: $f(x) \simeq x \leftarrow$

With C_2 it is possible to ground the request variable x in C_1 to the term $f(q)$, triggering a successful access to C_1^{ext} and a subsequent refutation. However, there is also this alternative derivation: First one may choose to apply C_3 to C_2 in a unit-sup-right inference, deriving the new unit $C_4 = p(q)$. This unit is smaller than its parent C_2 and makes it redundant, so C_2 is removed. This means that now the request variable x in C_1 can no longer be instantiated to $f(q)$, only to q by C_4. Since q is not an acceptable request term in the external source, no access and no refutation are possible. If C_3 could be used to similarly rewrite C_1^{ext} to $ext(q, a)$, then the problem would be solved, but the limited access to external sources prevents this. Thus we see that if a web service only accepts a more complex input even though we know of an equivalent less complex term, then we still have to provide the complex input to obtain a response. There does not appear to be a good solution to this. The example shows that forbidding destructive rewriting of *ext*-literals does not help, as it was the rewriting of an ordinary literal which caused the problem. Preserving the redundant C_2 after the derivation of C_4 would help, but this calls our notion of redundancy into question. A modified version of redundancy would have to require that beyond the usual criteria the terms of a redundant clause must not be able to take part in successful requests to external sources, a condition that is practically impossible to test. Redundancy elimination is such a powerful tool in ATP that any such restriction of redundancy would severely hamper the performance of provers. The modified range-restriction from the previous section can only provide a solution if rewriting of domain clauses is forbidden, because otherwise an equation like C_3 can simplify even the domain enumerating units, preventing the required complex request term from ever being derived.

An unpleasant aspect of this problem is the way the sequence of inference applications affects the outcome, as we are clearly losing proof confluence here. This lessens the usefulness of the notion of response-completeness, since even a fair strategy may obtain more or less responses from the external source, depending on the order of the inferences applied.

On a more positive note this situation may not be very common in actual QA usage, as the input accepted by web services is unlikely to have a form that could be simplified. Synonyms come to mind as a potential counter example, but successful web services should be sufficiently robust to provide the same response to synonymous requests.

The other problem is caused by subsumption in its various forms. A non-ground term may subsume a ground term and thereby prevent the ground term from serving as a request to the external source. Consider this example:

Example 16.8.

C_1^{ext}**:** $ext(q, a) \leftarrow$

— — — — — — — — — — — — — — — — — — — —

C_1**:** $\leftarrow ext(x, y), p(x)$

C_2**:** $p(q) \leftarrow$

C_3**:** $p(x) \leftarrow$

Here C_2 can instantiate the request variable x in C_1, which in turn triggers a successful request to C_1^{ext} and a subsequent refutation. If on the other hand C_3 is first used to subsume C_2, then we lose the ability to access C_1^{ext}, and no refutation is possible. Again we lose proof confluence, and again forbidding the offending operation only for *ext*-literals is no remedy, as it is an ordinary literal being subsumed. Instead subsumption would have to be forbidden entirely, a drastic measure with a severe impact on prover performance. Our modified range-restriction can solve this problem, as the ground request term q can then still be obtained from the domain clauses, even after the deletion of C_2. However, the negative effects of this clause transformation must be kept in mind.

Conclusions regarding Incompleteness

Problems such as these described are probably unavoidable consequences of the technical limitations of web services and the need to allow the dynamic generation of requests during the reasoning via variables. These reasons are independent of the calculus and the FOL representation of external sources. As such the problems we identified can be found elsewhere. For instance, while the SPASS-XDB system addresses some issues contributing to incompleteness [SST+10], we have found its completeness and proof confluence can still be negated by rewriting and subsumption, as in the Examples 16.7 and 16.8. Here is an example demonstrating the rewriting problem on SPASS-XDB, using the compatible TPTP syntax:

F_1**:**
```
fof(f1,axiom,(m='Moscow')).
```

F_2**:**
```
fof(f2,conjecture,?[Lat,Long,Name,Country]:
    latlong('Moscow',Lat,Long,Name,Country)).
```

The predicate `latlong/5` is used to access an external source for data about cities. Its first argument must be instantiated by a city name, and after a successful request the responses instantiate the other variables. Without F_1 SPASS-XDB finds a proof, but with F_1 the occurrence of `'Moscow'` in F_2 is rewritten to `m`; this prevents a successful retrieval, and the system gives up.[5]

[5]Note that the order of the formulas F_1 and F_2 is important here to demonstrate the problem, as it ensures that `m` counts as smaller than `'Moscow'` in the symbol precedence.

An analogous example of the subsumption problem is:

F_1: `fof(f1,conjecture,?[City,Lat,Long,Name,Country]:`
`(city(City) & latlong(City,Lat,Long,Name,Country))).`

F_2: `fof(f2,axiom,(city('Moscow'))).`

F_3: `fof(f3,axiom,(city(X))).`

As F_3 subsumes F_2, the proof becomes impossible, and the system gives up.

While it may not be possible to prevent such problems in the calculus, in practice such situations can be avoided by careful construction of the clauses, in particular those making use of external sources, and by choosing a suitable encoding of the knowledge. For example, the axiom F_3 above essentially states that everything is a city - such all-subsuming axioms do not make much sense in a real knowledge base. Likewise, rewriting request terms will in practice only be done when the result is semantically equivalent. Replacing `'Moscow'` with the abstract identifier `m` as above is hardly useful in the real world. Replacing it with a synonym or translation (like the Russian `'Moskva'`) can make sense, for example when normalizing all synonyms to one canonical form, but then the external source can likely handle the replacement as well (as is the case with `'Moskva'`).

The modified range-restriction can make systems with external sources "more complete", as can the treatment of responses as input clauses. Both methods may have a negative impact on the performance, though. Finally, we see no solution to problems such as Example 16.1, although clauses like its $C_1 = \ \leftarrow ext(x, y)$ make little sense in practice. Completeness may be impossible when it comes to the integration of external sources, but we believe this is not a severe problem in an actual application.

16.2 Implementation

Our implementation consists of two parts: the modifications to E-KRHyper and a separate module that provides an interface between the prover and the web services. The interface module was developed by Markus Bender according to my specifications. It is responsible for the actual web service access, and it enables an asynchronous delivery of the responses to E-KRHyper, allowing the prover to reason concurrently while the interface carries out the slow web service communication, collecting the responses from the external sources in a cache. The interface module accepts request terms from E-KRHyper over a socket. Each such request is immediately compared against the response cache and then answered synchronously[6] with one of the following three reply types:

***wait*:** There is no response to this request in the cache yet. This may be due to one of two possible reasons, the first of which is that this may be the first time E-KRHyper has sent this request to the interface. In that case, after sending the *wait* reply the interface transforms the respective request subterms into a compatible query, taking into account capitalization, number

[6]Note that the communication between prover and interface is synchronous, but the delivery of a web service response is asynchronous to the initial request.

format and so on. Then the interface determines the web service to be addressed and forwards the compatible request over the web. The other possible reason for sending the *wait* reply is that E-KRHyper may already have sent this request to the interface module before, but the interface has not yet received a response from the web service. Either way the *wait* reply tells the prover that it can continue its reasoning with other inferences for now, but also that it should check for a response later on, as the web service communication is still in progress.

***failed*:** The interface module has already concluded the web service communication regarding this particular request, but it received a failed response. This may be because the web service is currently offline, or because it does not have any information to answer that request. From a FOL point of view this is treated as the external source not containing any instance of $ext(q, a)$ for the request q and the possibly non-ground response term a.

***<response>*:** The interface module has already concluded the web service communication regarding this particular request. It has received a proper response and it has converted this into a response term compatible with E-KRHyper. This response term has been stored in the cache of the interface module, and it is now sent to E-KRHyper as a reply.

E-KRHyper makes web service requests during the normal inferencing within its lower level algorithm, highlighted in the pseudo-code Algorithm 16.1. To minimize access times the prover maintains its own additional cache of requests and responses. When making a request, E-KRHyper first checks whether there is already a response for this request in its own cache or the request has been marked as failed, and only if neither holds does the prover send the request to the interface module. The prover also keeps track of which requests are still waiting for a response, indicated by the *pending* set. The default setting of E-KRHyper is to keep looping in its lower level algorithm if there are no more inference possibilities in the current branch, except for requests still waiting for responses. As an alternative it is possible to have the prover postpone such branches. In that case E-KRHyper will continue with a different branch. A postponed branch is revisited when E-KRHyper has to select a new branch because the current branch has been closed or exhausted, and either at least one of the pending queries in the postponed branch has been answered, thereby opening new inference possibilities, or all pending queries there have been marked as *failed*, meaning the branch is truly exhausted.

The interface module is unaware of when E-KRHyper finishes a derivation and when the prover starts a new proof attempt. As we assume external data to be atemporal, the interface module preserves its cache between derivations unless it is restarted, and E-KRHyper can use cached responses that were requested during earlier proof attempts.

Algorithm 16.1 The lower level algorithm in E-KRHyper with web service access highlighted

function EVALUATE_BRANCH_AT_NODE(*node*)
  if *node* = *rootNode* **then**
    clauses := input reasoning problem;
    clauses := reduce *clauses* by *clauses*;
    conclusions := all inferences with *clauses*
      and cached web service responses;
    ***pending* := (*pending* \ (newly answered or failed requests))**
      ∪ (new requests while computing *conclusions*);
    overweight := inference results above *weightLimit*;
    conclusions := *conclusions* \ *overweight*
  else
    conclusions := {split literal labeling *node*}
  end if
  conclusions := reduce *conclusions* by *clauses*;
  while (*conclusions* ∪ ***pending*** ≠ {}) ∧ (contradiction not found) **do**
    clauses := reduce *clauses* by *conclusions*;
    disjunctions := *disjunctions* ∪ (disjunctions from *conclusions*);
    newClauses := *conclusions* \ *disjunctions*;
    conclusions := all inferences with *newClauses*
      and cached web service responses,
      using other premises from *clauses*;
    ***pending* := (*pending* \ (newly answered or failed requests))**
      ∪ (new requests while computing *conclusions*);
    overweight := *overweight* ∪ (inference results above *weightLimit*);
    conclusions := *conclusions* \ *overweight*;
    clauses := *clauses* ∪ *newClauses*;
  end while
  if contradiction found **then**
    return *closed*
  else
    return *exhausted*
  end if
end function

The web services currently supported are:

***ECB Currency Exchange Rates*:** Currency exchange rates for major currencies can be retrieved from the European Central Bank[7], where they are updated daily.

***Yahoo! GeoPlanet*:** The *Yahoo! GeoPlanet*[8] web service offers data related to geographical place names. Given the name of a city, a country, a landmark or similar, our interface retrieves latitude, longitude, and a short string indicating whether the name refers to a city, a country and so on. It is also possible to retrieve information about the geographical hierarchies, for example to find out in which country a given city is located.

[7] `http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml`
[8] `http://developer.yahoo.com/geo/geoplanet`

Multiple responses are possible, since the same name can be used for different geographical places.

***LogAnswer Ontology Browser*:** An experimental online database developed by the IICS to expand upon the MultiNet knowledge, the *LogAnswer Ontology Browser*[9] stores and links data from other ontologies. The initial source was OpenCyc, and since then the browser has been expanded to include data from the German Wikipedia and the English DBpedia. For the time being this data is mostly confined to concepts expressed by nouns, and the hierarchy between these is represented mostly by *same_as* and *subclass_of* relations. By linking English and German nouns it will become possible for the German LogAnswer to make use of English language concept hierarchies. This integration of the Ontology Browser is still future work, though. However, E-KRHyper and the web service interface module support access to this online database, and the upcoming Section 16.3 will describe a special usage of these ontologies for the purpose of relaxation.

***System Q&A TPTP*:** This intermediate web service[10] provides access to the external sources used by the SPASS-XDB system. By connecting to this web service E-KRHyper can gain access to many of these sources, too. However, this is highly experimental and unreliable, as System Q&A TPTP requires requests in the form of formulas in TPTP syntax, whereas E-KRHyper accesses web services over the request terms within *ext*-literals. Therefore reasoning problems for E-KRHyper would have to encode full TPTP formulas as terms within *ext*-literals, a cumbersome undertaking that is prone to bugs due to it blurring the line between predicate and function symbols. In the long run it would make more sense to access the external sources directly, in particular since the current set-up is inefficient in that it puts two interfaces between the prover and the actual web services.

Overall it can be said that the integration of web services in E-KRHyper is still experimental. The range of accessible services is limited, and for an actual usage within LogAnswer it would be necessary to expand the background knowledge base by rules which make use of the external data and put it into the context of the MultiNet ontology. This is future work for a knowledge engineer, and it lies outside the scope of this AR-centric dissertation. However, the current implementation is a proof of concept showing that the approach is feasible, and it has a sound formal basis in the calculus extensions which take into account the inherent technical limitations of external sources.

[9] `http://www.loganswer.de/hop/loganswer-cyc`
[10] `http://www.cs.miami.edu/~tptp/cgi-bin/SystemQATPTP`

16.3 Abductive Relaxation

The relaxation method in LogAnswer relies on a dropping of literals from the query clause, see Section 13.1. Naturally, this is a relatively crude method. The heuristics based on reasoning and semantics try to ensure that the literal selection improves the provability of the query without distorting its meaning. Nevertheless the approach also relies on the hope that the FOL representation is sufficiently fine-grained so that each single literal is not overly important, and its removal has little semantic impact. This can be risky. For example, asking LogAnswer the question *"What is the weight of the 'Maus' ('Mouse') tank?"*[11] will result in the gram weights of various species of mice, rather than the 188 tons of the German tank prototype from World War II. Obviously *"tank"* is a critical piece of information here that should not have been skipped, though at least LogAnswer states low confidence values for its answers, with none higher than 14%.

Rather than completely dropping literals, it would be preferable to replace them with semantically related, more general literals. The LogAnswer Ontology Browser mentioned in the previous section provides access to a wealth of hierarchical concept information. As mentioned, much of this is expressed as *subclass_of* relationships between concept identifiers, where *subclass_of*(c, d) expresses that any entity of concept c also belongs to concept d, for example *subclass_of*(*tank*, *vehicle*). In a simplistic FOL representation where predicates are used as concepts, such a subclass relationship could be expressed as the formula $\forall x(c(x) \rightarrow d(x))$. When concepts are represented as constant terms, then a formula like $\forall x(is\_a(x, c) \rightarrow is\_a(x, d))$ serves the same purpose. Given an entity of concept c it is then trivial to deduce that it also belongs to d. The opposite direction is *abduction*: Given an entity of concept d, one can abduce that it may also belong to c. Abduction is not sound, its result is a hypothesis, an assumption. However, we may see now how it can help in relaxing a query. Consider the question from above together with the candidate passage C:

Q: *"What is the weight of the 'Maus' ('Mouse') tank?"*

C: *"At 188 tons the 'Maus' is the heaviest armoured fighting vehicle ever built."*

As tanks are a subclass of vehicles, given C we can use abduction to form the hypothesis that the vehicle *'Maus'* is a tank and then answer the question.

We will now describe a clause set transformation which aims at using an external source of axioms expressing a concept hierarchy for the purpose of query relaxation guided by abduction. This transformation is implemented in E-KRHyper, where it works in combination with the web service interface module and the LogAnswer Ontology Browser. The specifics of this transformation are influenced by the following considerations. In order to reuse as much input between proof attempts as possible, we minimize the number of new clauses. The abductive relaxation affects primarily the query clause, whereas clauses from the knowledge base remain untouched. In the example above one could imagine an approach where the FOL representation of the candidate passage C is replaced by a more specific one, basically working with the passage *"At 188 tons the 'Maus' is the heaviest armoured fighting tank ever built."* This might be closer to actual abduction, but as it is impractical for our purposes,

[11] *"Wieviel wiegt der Panzer 'Maus'?"*

instead we relax the query by using a more general term, i.e. *"What is the weight of the 'Maus' ('Mouse') vehicle?"* The result is effectively the same, and the query term replacement can also be seen as abductive because given an answer to the relaxed query, an answer to the original query could be formed by abduction. In other words, we hypothesize that an answer to the relaxed query is relevant for the original as well. Another consideration is that due to the inherent uncertainty of abduction, the user should receive not only answers, but also hints as to what abductive assumptions LogAnswer made, so that the user can judge whether the answer is applicable.

We first describe the basic transformation. A few possible modifications are discussed afterwards. Let $\mathcal{C}$ be a set of clauses with a negative query clause $Q = \leftarrow Q_1, \ldots, Q_n$ with $n \geq 0$. Let $\mathcal{C}^{ext}$ be an external source containing positive ground *ext*-units of the form $ext(subclass\_of(c), d) \leftarrow$, which is the external source conforming representation of the subclass relationship $subclass\_of(c, d)$ between two concept identifiers c and d. We obtain the abductive relaxation supporting clause set $\mathcal{C}^{ar}$ from $\mathcal{C}$ by adding two clauses as follows.

First, add Q^{ar} with

$$Q^{ar}\text{:}\ relaxed\_answer(rlx(c_1, x_1), \ldots, rlx(c_m, x_m)) \leftarrow Q'_1, \ldots, Q'_n, ext(subclass\_of(c_1), x_1), \ldots, ext(subclass\_of(c_m), x_m)$$

where $c_1, \ldots, c_m$ ($m \geq 0$) are the occurrences of constants in $Q_1, \ldots, Q_n$, and where $Q'_1, \ldots, Q'_n$ are obtained from $Q_1, \ldots, Q_n$ by replacing each c_i ($0 \leq i \leq m$) with a fresh variable x_i. *relaxed_answer* is a new predicate symbol of arity m, and *rlx* is a new binary function symbol.

Secondly, add a unit clause C^{rs} expressing the trivial reflexive subclass relationship of a concept with itself:

$$C^{rs}\text{:}\ ext(subclass\_of(x), x) \leftarrow$$

As $\mathcal{C} \subset \mathcal{C}^{ar}$, any refutational proof and answer derivable for $\mathcal{C}$ can also be derived for $\mathcal{C}^{ar}$. The intention behind Q^{ar} is as follows. By moving the concept identifiers $c_1, \ldots, c_m$ out of the original query literals $Q_1, \ldots, Q_n$ into the new *ext*-literals and replacing their original occurrences with the response variables, it becomes possible to request more general superclass concepts from the external source and to insert these into the query. As only constants are treated this way, all the new *ext*-literals have ground request terms, making them valid for accessing the external source. The trivial reflexive subclass unit ensures that concepts do not have to be relaxed if they can already be proven without external access. Finally, once all negative literals of Q^{ar} have been refuted with an overall substitution σ, the derived unit $relaxed\_answer(rlx(c_1, x_1), \ldots, rlx(c_m, x_m))\sigma \leftarrow$ provides information about which concepts were relaxed by which more general concepts. As Q^{ar} cannot be used to close a branch, any *relaxed_answer* units derived from this clause function as a fallback in a QA situation: If E-KRHyper does not find a refutational proof for Q within the allotted time, it can return the *relaxed_answer* units found in the branch instead, leaving it to the overarching LogAnswer system or the user to decide whether the generalizations are acceptable.

An example will illustrate the principle:

C_1^{ext}: $ext(subclass\_of(tank), vehicle) \leftarrow$

Q: $\leftarrow is\_a('Maus', tank), has\_weight('Maus', x)$

C_1: $is\_a('Maus', vehicle) \leftarrow$

C_2: $has\_weight('Maus', '188t') \leftarrow$

Q^{ar}: $relaxed\_answer(rlx('Maus', x_1), rlx(tank, x_2), rlx('Maus', x_3)) \leftarrow$
$is\_a(x_1, x_2),$
$has\_weight(x_3, x),$
$ext(subclass\_of('Maus'), x_1),$
$ext(subclass\_of(tank), x_2),$
$ext(subclass\_of('Maus'), x_3)$

C^{rs}: $ext(subclass\_of(x), x) \leftarrow$

The original query Q, specifically its first literal, cannot be proven in this set of clauses. However, the relaxation query Q^{ar} can: Its first body literal atom $is\_a(x_1, x_2)$ unifies with C_1, instantiating x_1 with *'Maus'* and x_2 with *vehicle*. The second body literal atom $has\_weight(x_3, x)$ unifies with C_2, instantiating x_3 with *'Maus'* and x with *'188t'*. Then there are the *ext*-literals to deal with. While the external source in the example contains no subclass information for *'Maus'*, the first and the third *ext*-atom, both instantiated by the above substitutions to $ext(subclass\_of('Maus'), 'Maus')$, unify with the trivial subclass unit C^{rs}. The second *ext*-atom on the other hand has been instantiated to $ext(subclass\_of(tank), vehicle)$, which does not unify with C^{rs}. It is a valid request to the external source, though, and the response term *vehicle* from C_1^{ext} matches the already instantiated response term in Q^{ar}, thus proving the final body literal. We derive a positive literal or unit clause C_3:

$$C_3 : relaxed\_answer(rlx('Maus', 'Maus'), rlx(tank, vehicle), rlx('Maus', 'Maus')) \leftarrow$$

This indicates that a proof is possible if we accept generalizing *tank* to *vehicle*. The other two "generalizations" are trivial, and we ignore them. In a QA system like LogAnswer this information could be used to answer the question *"What is the weight of the 'Maus' tank?"* with *"188t, if by 'tank' you mean 'vehicle"'*.

Several modifications are thinkable:

- Instead of allowing the relaxation of all constants in Q, it may make sense to leave some of these terms in their literals instead of replacing them with response variables from new *ext*-literals. This may be the case for constants representing words which the shallow retrieval phase already found to match between question and candidate passage, or if some word is considered to be so critical that it should not be relaxed.

- When using a *FOCUS* variable as in LogAnswer it may be helpful to use *relaxed_answer* as a predicate with the arity $m+1$ instead of just m, and to include *FOCUS* as an additional subterm. This makes it easier to extract the specific answer from a *relaxed_answer* unit.

- In principle there is no need to restrict the relaxations to generalizations determined by *subclass_of* relationships. The LogAnswer Ontology Browser also stores many concept equivalences via the *same_as* relation, and these could similarly relax a question by broadening its scope. Even more specific concepts accessible via a *superclass_of* relation could help this way; as long as the transformation allows the query to match more terms, it can be regarded as a form of relaxation.

- If saving time is of paramount importance and any proof is good enough, then a negative unit $\neg relaxed\_answer(x_1, \ldots, x_m)$ added to $\mathcal{C}^{ar}$ can ensure that even *relaxed_answer* units can close the current branch, thereby terminating any further computation there.

- Our abductive relaxation can be combined with the currently implemented skipping relaxation, as Q'^{ar} can obviously also be formed for a conventionally relaxed query clause Q' obtained by dropping a literal from Q.

Query relaxation by abduction has been researched before by Terry Gaasterland et al. [GGM92], but their approach differs in that it assumes predicates to express concepts and implications to express subclass relations. The method requires top-down reasoning and is only intended for Horn-clauses. It assumes full accessibility to the knowledge base by the reasoning system, which is unfeasible in an open-domain QA system. It also results in a potentially large number of additional clauses by transforming the entire knowledge base, adding for every clause $A \leftarrow B, \mathcal{B}$ its *reciprocal clause* $B' \leftarrow A, \mathcal{B}$, effectively turning the implication around and marking the new head literal B' as derived by abduction. Thus it may be more flexible in that it allows a wider range of abductive relaxations - when $\mathcal{B}$ above is not empty the clause can express more than just a relation between two concepts - but such more complex steps will be difficult to explain to the user, who should be the final judge over the acceptability of relaxation. Finally, our approach will be more efficient at least in the LogAnswer setting in that the original query and the relaxed query are treated in the same proof attempt, whereas the method above is intended to be applied iteratively, starting with the original query and then relaxing it in the case of failure.

Chapter 17

Conclusions and Future Work

In this dissertation we have explored the usage of automated reasoning, in particular automated theorem proving, in the context of question answering. Let us briefly review the main points.

Starting out with the fundamental differences between ATP and the natural language processing methods of QA, namely precision versus robustness and deep reasoning versus fast, shallow information retrieval, we have identified how an integration of an automated theorem prover into a QA system can lead to an improvement of QA, but also which obstacles need to be overcome for this. We have implemented such an integration by embedding the theorem prover E-KRHyper into the QA system LogAnswer. Focusing on E-KRHyper we have described the theoretical background and the implementation of the prover. After a general overview of LogAnswer we have detailed the adaptation of E-KRHyper to its embedding in this QA system. First we have shown how the basic architecture and strategy of a prover can have an effect on QA performance by comparing E-KRHyper with its relative E-Darwin on LogAnswer reasoning problems. Then we have gone into the details of the implementation which serve to further enhance the performance of E-KRHyper on the reasoning tasks expected in LogAnswer. This includes the indexing of multi-literal clauses, for which we have devised a method that remains effective for backward subsumption. Several modifications turn E-KRHyper into a reasoning server that remains in continuous operation while carrying out all proof attempts for LogAnswer. Axiom selection is a useful approach to make large reasoning problems easier to solve, and we discuss complete and incomplete techniques and their implementation in E-KRHyper. To overcome the brittleness of precise ATP we have introduced relaxation support to enhance the robustness of E-KRHyper. For most of the described implementational aspects we have done separate evaluations to show their effectiveness, including a comparison with the original KRHyper demonstrating that the general theorem proving capability of E-KRHyper has not been diminished - on the contrary, E-KRHyper outperforms its parent by a significant margin. We have also summarized the performance of E-KRHyper in the annual CASC competition. Not all modifications can be properly tested with reasoning problems like the TPTP, though,

and instead they require the full LogAnswer system. For such evaluations we have participated with LogAnswer in various CLEF QA competitions over the years. We have analyzed the results of these, not only to compare the performance of LogAnswer to other QA systems, but also to identify weaknesses of our system in order to improve it.

Finally, we have considered two experimental ways to broaden the scope and appeal of QA. The first is to participate with LogAnswer in online QA forums, where our system could disburden human users from answering factoid questions, and where the visitors in turn provide us with a wealth of questions, resulting in a large scale real-world evaluation of LogAnswer. The second is to connect LogAnswer to external sources like web services which provide current data that goes beyond static encyclopedic knowledge bases. To put this proposition on a sound formal basis we have extended the hyper tableaux calculi underlying E-KRHyper and implemented these extensions in the prover. A special application of this is to use the concept hierarchies of external ontologies to relax questions by abduction rather than by skipping of literals. We have presented a special clause transformation for this purpose, which is also implemented in E-KRHyper.

This dissertation has shown that automated reasoning can be a useful tool in QA, provided that the theorem prover is embedded in a manner that takes into account the brittleness resulting from the precision of logic and the short time limits imposed by having to test a large amount of answer candidates within an acceptable response time. To achieve this the prover must forgo the conventional ATP approach of working only on one reasoning problem and trying to fully solve it. Rather, partial results achieved in a short time may be preferable to a full solution that takes longer to reach. For this the prover needs the flexibility to restart a proof attempt with only minor changes to the input without having to repeat much previous work, and its partial results should be able to guide the overarching relaxation control.

In the CLEF competitions LogAnswer was atypical among the participants for its use of automated reasoning, yet our system was competitive once the initial teething troubles had been ironed out, achieving results above the average. While unfortunately the competition design in later years prevented logic from contributing much to the actual answers, the logic-based proof features have shown to be a powerful criterion in deciding when to reject answers, and LogAnswer's good ability to recognize when not to answer has been remarked upon by the CLEF organizers on several occasions. Our own evaluation on questions from a QA forum has shown that this ability will be useful and essential in a real-world application of QA. Further improvements are required for actual usage, though, as questions by normal users have proven to be vastly more difficult to handle automatically than the curated questions in the QA competitions.

Plenty of work remains to be done in QA research before such systems can gather the widespread acceptance of search engines and become everyday tools. For LogAnswer in particular, apart from general performance improvements, the following specific areas must be addressed: The background knowledge base should be extended to further exploit the reasoning capabilities of E-KRHyper, for example by non-Horn clauses, more arithmetics and equality, and by making use of the web services accessible by the prover. This would mean that the rich concept hierarchies of external ontologies could be utilized for question answering, rather than remaining limited to test examples for E-KRHyper on

its own. With improvements to LogAnswer the system may eventually become sufficiently reliable for an actual integration in a QA forum. This would finally provide a most desirable large scale evaluation of LogAnswer by real users.

A fairly obvious way to broaden the scope of LogAnswer in practice would be to support the English language. Many components of the system are language-independent, but nevertheless such an extension would require significant changes to the parser, the shallow retrieval phase and the answer generation. While no qualitative improvement in QA is to be expected as both languages have a similar expressivity, an English LogAnswer could attract more users than the current German version and allow a more widespread integration.

An improvement in QA quality can be expected from investments in the hardware infrastructure. By exploiting parallelization (see Section 14.5) LogAnswer could perform deeper reasoning on more answer candidates in less time. Unfortunately the costs of such an endeavour are difficult to handle in a research project with the current size of LogAnswer, and they may confine such improvements to commercial applications.

As for E-KRHyper, performance improvements to the prover may directly benefit LogAnswer as well, although this must be approached with caution. The basic strategy of E-KRHyper has proven to be very effective on LogAnswer reasoning problems, yet on general TPTP problems as encountered in CASC this very same strategy poses a severe limitation that likely forms a performance ceiling which will prevent the prover from reaching the higher ranks. The strength of top-ranked systems like Vampire comes in part from their ability to adapt their strategy to the class of the given problem. The semi-naive evaluation strategy of E-KRHyper is uniform for all problem classes, and its rigid breadth-first method offers little room for class-specific adaptations. The popular given-clause algorithm allows heavy use of heuristics when selecting a clause for inferencing, a point where adaptations can have a great effect. E-Darwin, our own given-clause algorithm prover, has exhibited large performance differences on the same problem depending on experimental changes in its selection heuristics. The side-project status of this prover has prevented it from receiving the attention and evaluation necessary to turn its own uniform strategy into an adaptive one. Hence for general theorem proving and CASC E-Darwin is a more promising contender than E-KRHyper, as the former has the potential for more flexibility in its strategy.

Both E-Darwin and E-KRHyper are involved in various areas of research under investigation in the AGKI. Axiom selection strategies are a promising field which can find more use in LogAnswer, and the AGKI is experimenting with graph-based partitioning and with clustering methods in order to handle large knowledge bases. Another such area is qualitative spatial-temporal reasoning, which could help LogAnswer in evaluating prepositions and other means of expressing spatial and temporal relationships. E-KRHyper has been used in experiments with formalisms such as Allen's interval algebra [All83], and the IICS is researching an extension of MultiNet with such features. A third area of major experimentation concerns description logics (DL), which commonly find application in knowledge representation and semantic networks. The E-hyper tableaux calculus has been modified to support reasoning on $\mathcal{SHIQ}$, enabling E-KRHyper to work on DL-ontologies like *Galen*.[1]

[1] `http://www.opengalen.org`

As indicated by the breadth of the possible extensions we discussed, automated QA is a field that can merge a wide range of AI techniques in an application with a single purpose, the answering of questions in natural language. We hope that this dissertation contributes to the integration of automated theorem proving into QA, and that it provides useful suggestions to further extensions.

Curriculum Vitae

Personal Information

Name:	Björn Pelzer
Date/Place of Birth:	22 October 1976, Koblenz, Germany
Nationality:	German and Swedish
Address:	Yorckstraße 29, 56073 Koblenz, Germany
Phone:	(+49) 261 287 2776
Email:	bpelzer@uni-koblenz.de

Education

Schools:	
08/1983 - 07/1987	elementary school: Grundschule Buchholz
09/1987 - 06/1996	grammar school: Kantgymnasium Boppard (Abitur)
Military Service:	
07/1996 - 04/1997	Idar-Oberstein, artillery surveyor and reconnaissance driver at Beobachtungspanzerartillerielehrbataillon 51
University:	
10/1997 - 03/2007	computer science at Universität Koblenz-Landau: • 1997 - 2001: specialization: computational linguistics • 2001 - 2007: specialization: artificial intelligence, secondary subject: philosophy • qualification: Diplom-Informatiker

Professional Career

04/2005 - 03/2007	student assistant in the AI Research Group AGKI at Universität Koblenz-Landau
08/2007 - 04/2012 and again since 09/2012	research assistant in the AGKI: • researcher in the LogAnswer project • tutor in logic, AI and programming (Prolog) • supervisor for student theses and seminaries • organizer for the student exchange programs with Chalmers University of Technology (Gothenburg, Sweden) and Osaka University (Osaka, Japan)

Own Publications

[1] Peter Baumgartner, Ulrich Furbach, and Björn Pelzer. Hyper Tableaux with Equality. In *Automated Deduction - CADE-21, Proceedings*, 2007.

[2] Peter Baumgartner, Ulrich Furbach, and Björn Pelzer. The Hyper Tableaux Calculus with Equality and an Application to Finite Model Computation. *Journal of Logic and Computation*, 20(1):77–109, 2010.

[3] Peter Baumgartner, Björn Pelzer, and Cesare Tinelli. Model Evolution with Equality - Revised and Implemented. *Journal of Symbolic Computation*, 47(9):1011 – 1045, 2012.

[4] Tiansi Dong, Ulrich Furbach, Ingo Glöckner, and Björn Pelzer. A Natural Language Question Answering System as a Participant in Human Q&A Portals. In Toby Walsh, editor, *IJCAI*, pages 2430–2435. IJCAI/AAAI, 2011.

[5] Tiansi Dong, Ingo Glöckner, and Björn Pelzer. LogAnswer in Question Answering Forums. In Joaquim Filipe and Ana L. N. Fred, editors, *ICAART (1)*, pages 492–497. SciTePress, 2011.

[6] Ulrich Furbach, Ingo Glöckner, Hermann Helbig, and Björn Pelzer. LogAnswer - A Deduction-based Question Answering System. In *Automated Reasoning (IJCAR 2008)*, Lecture Notes in Computer Science, pages 139–146. Springer, 2008.

[7] Ulrich Furbach, Ingo Glöckner, Hermann Helbig, and Björn Pelzer. Logic-Based Question Answering. *KI*, 24(1):51–55, 2010.

[8] Ulrich Furbach, Ingo Glöckner, and Björn Pelzer. An Application of Automated Reasoning in Natural Language Question Answering. *AI Communications*, 23(2-3):241–265, 2010. PAAR Special Issue.

[9] Ingo Glöckner and Björn Pelzer. Combining Logic and Machine Learning for Answering Questions. In *Evaluating Systems for Multilingual and Multimodal Information Access, 9th Workshop of the Cross-Language Evaluation Forum, CLEF 2008, Aarhus, Denmark, September 17-19, 2008, Revised Selected Papers*, pages 401–408, 2008.

[10] Ingo Glöckner and Björn Pelzer. Exploring Robustness Enhancements for Logic-Based Passage Filtering. In *Knowledge Based Intelligent Information and Engineering Systems (Proceedings of KES2008, Part I)*, LNAI 5117, pages 606–614. Springer, 2008.

[11] Ingo Glöckner and Björn Pelzer. Extending a Logic-Based Question Answering System for Administrative Texts. In Carol Peters, Giorgio Maria Di Nunzio, Mikko Kurimo, Djamel Mostefa, Anselmo Peñas, and Giovanna Roda, editors, *CLEF (1)*, volume 6241 of *Lecture Notes in Computer Science*, pages 265–272. Springer, 2009.

[12] Ingo Glöckner and Björn Pelzer. The LogAnswer Project at ResPubliQA 2010. In Martin Braschler, Donna Harman, and Emanuele Pianta, editors, *CLEF (Notebook Papers/LABs/Workshops)*, 2010.

[13] Ingo Glöckner, Björn Pelzer, and Tiansi Dong. The LogAnswer Project at QA4MRE 2011. In Vivien Petras, Pamela Forner, and Paul D. Clough, editors, *CLEF (Notebook Papers/Labs/Workshop)*, 2011.

[14] Björn Pelzer. Adding Equality Reasoning to the KRHyper Theorem Prover. Diploma thesis, Universität Koblenz-Landau, 2007.

[15] Björn Pelzer and Ingo Glöckner. Combining Theorem Proving with Natural Language Processing. In *Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning (PAAR-2008/ESHOL-2008), Sydney, Australia, August 10–11, 2008*, pages 71–80. CEUR Workshop Proceedings, 2008.

[16] Björn Pelzer and Christoph Wernhard. System Description: E-KRHyper. In *Automated Deduction - CADE-21, Proceedings*, pages 508–513, 2007.

References

[All83] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[Baa03] Franz Baader, editor. *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*. Springer, 2003. ISBN 3-540-40559-3.

[BB04] Peter Baumgartner and Aljoscha Burchardt. Logic Programming Infrastructure for Inferences on FrameNet. In José Alferes and João Leite, editors, *Logics in Artificial Intelligence, Ninth European Conference, JELIA'04*, volume 3229, pages 591–603. Springer, 2004.

[BCC+03] John Burger, Claire Cardie, Vinay Chaudhri, Robert Gaizauskas, Sanda Harabagiu, David Israel, Christian Jacquemin, Chin-Yew Lin, Steve Maiorano, George Miller, Dan Moldovan, Bill Ogden, John Prager, Ellen Riloff, Amit Singhal, Rohini Shrihari, Tomek Strzalkowski, Ellen Voorhees, and Ralph Weishedel. Issues, Tasks and Program Structures to Roadmap Research in Question Answering. Technical report, SRI International, 2003.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 151–165, Yerevan, Armenia, October 2007. Springer.

[BF05] Peter Baumgartner and Ulrich Furbach. Living Books, Automated Deduction and Other Strange Things. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 249–267. Springer, 2005. ISBN 3-540-25051-4.

[BFGH+03] Peter Baumgartner, Ulrich Furbach, Margret Gross-Hardt, Thomas Kleemann, and Christoph Wernhard. KRHyper In-

side - Model Based Deduction in Applications. Fachberichte Informatik 15–2003, Universität Koblenz-Landau, 2003.

[BFGHK04] Peter Baumgartner, Ulrich Furbach, Margret Gross-Hardt, and Thomas Kleemann. Model-based Deduction for Database Schema Reasoning. In Susanne Biundo, Thom Frühwirth, and Günther Palm, editors, *KI 2004: Advances in Artificial Intelligence*, volume 3238, pages 168–182. Springer, 2004.

[BFGHS04] Peter Baumgartner, Ulrich Furbach, Margret Gross-Hardt, and Alex Sinner. Living Book – Deduction, Slicing, and Interaction. *Journal of Automated Reasoning*, 32(3):259–286, 2004.

[BFK94] Peter Baumgartner, Ulrich Furbach, and Universität Koblenz. PROTEIN: A PROver with a Theory Extension Interface. In *Automated Deduction – CADE-12, volume 814 of LNAI*, pages 769–773. Springer, 1994.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper Tableaux. In *JELIA'96, Proceedings*, pages 1–17, 1996.

[BFT04] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Darwin: A theorem prover for the model evolution calculus, 2004.

[BFT06] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the Model Evolution Calculus. *International Journal on Artificial Intelligence Tools*, 15(1):21–52, 2006.

[BG98] Leo Bachmair and Harald Ganzinger. Equational Reasoning in Saturation-based Theorem Proving. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume I, Foundations: Calculi and Methods.* Kluwer Academic Publishers, Dordrecht, 1998.

[Bib81] Wolfgang Bibel. Mating in Matrices. In *German Conference on Artificial Intelligence*, pages 171–187, 1981.

[Bla68] Fischer Black. A Deductive Question Answering System. In Marvin Minsky, editor, *Semantic Information Processing.* MIT Press, Cambridge, Massachusetts, 1968.

[Ble71] Woodrow Wilson Bledsoe. Splitting and Reduction Heuristics in Automatic Theorem Proving. *Artificial Intelligence*, 2(1): 55–77, 1971.

[BLK+09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A Crystallization Point for the Web of Data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[BM04] Peter Baumgartner and Anupam Mediratta. Improving Stable Models-based Planning by Bidirectional Search. In *International Conference on Knowledge Based Computer Systems (KBCS)*, Hyderabad, India, December 2004.

[Bos01] Johan Bos. DORIS 2001: Underspecification, Resolution and Inference for Discourse Representation Structures. In *ICoS-3 - Inference in Computational Semantics, Workshop Proceedings*, 2001.

[BP95] Bernhard Beckert and Joachim Posegga. leanTaP: Lean Tableau-based Deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[BPT08] Christoph Benzmüller, Lawrence C. Paulson, and Frank Theiss. LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In *Fourth International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNAI*. Springer, 2008.

[BS06] Peter Baumgartner and Fabian M. Suchanek. Automated Reasoning Support for First-Order Ontologies. In *Principles and Practice of Semantic Web Reasoning 4th International Workshop (PPSWR 2006), Revised Selected Papers*. Springer, 2006.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2010.

[BT03] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus. In Baader [Baa03], pages 350–364. ISBN 3-540-40559-3.

[BT05] Peter Baumgartner and Cesare Tinelli. The Model Evolution Calculus with Equality. In Nieuwenhuis [Nie05], pages 392–408. ISBN 3-540-28005-7.

[Bus45] Vannevar Bush. As We May Think. *Atlantic Monthly*, 176: 101–108, 1945.

[Cla03] Koen Claessen. New Techniques that Improve MACE-style Finite Model Finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[CMB05] Jon Curtis, Gavin Matthews, and David Baxter. On the Effective Use of Cyc in a Question Answering System. In *Proceedings of the IJCAI Workshop on Knowledge and Reasoning for Answering Questions (KRAQ'05)*, pages 61–70, 2005.

[CR93a] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. *SIGPLAN Notices*, 28:37–52, March 1993.

[CR93b] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4.

[DdM06] Bruno Dutertre and Leonardo de Moura. The Yices SMT Solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[Der82] Nachum Dershowitz. Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, 17:279–301, 1982.

[DKS96] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT - A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18:189–198, 1996.

[dNM06] Hans de Nivelle and Jia Meng. Geometric Resolution: A Proof Procedure Based on Finite Model Search. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2006. ISBN 3-540-37187-7.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, July 1960.

[EE68] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *Proceedings of AFIPS Fall Joint Computer Conference*, pages 395–410, San Francisco, California, December 1968.

[EM08] Levent Erkök and John Matthews. Using Yices as an Automated Solver in Isabelle/HOL. In *Automated Formal Methods 08*, pages 3–13. ACM Press, 2008.

[Eng62] Douglas C. Engelbart. Augmenting Human Intellect: A Conceptual Framework. Technical report, Stanford Research Institute, October 1962.

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-Solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. ISBN 3-540-20851-8.

[FBCC+10] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3):59–79, 2010.

[Fel98] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, first edition, 1998.

[FHB+97] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Judicaël Courant, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq Proof Assistant - Reference Manual Version 6.1. Technical report, INRIA, 1997.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[FPA+08] Pamela Forner, Anselmo Peñas, Eneko Agirre, Iñaki Alegria, Corina Forascu, Nicolas Moreau, Petya Osenova, Prokopis Prokopidis, Paulo Rocha, Bogdan Sacaleanu, Richard F. E. Sutcliffe, and Erik Tjong Kim Sang. Overview of the CLEF 2008 Multilingual Question Answering Track. In *CLEF*, pages 262–295, 2008.

[FSS99] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English - Not Just Another Logic Specification Language. In Pierre Flener, editor, *Logic-Based Program Synthesis and Transformation*, number 1559 in Lecture Notes in Computer Science, Manchester, UK, June 1999. Eighth International Workshop LOPSTR'98, Springer.

[Fur94] Ulrich Furbach. Theory Reasoning in First Order Calculi. In Kai von Luck and Heinz Marburger, editors, *IS/KI*, volume 777 of *Lecture Notes in Computer Science*, pages 139–156. Springer, 1994. ISBN 3-540-57802-1.

[GFH+07] Danilo Giampiccolo, Pamela Forner, Jesús Herrera, Anselmo Peñas, Christelle Ayache, Corina Forascu, Valentin Jijkoun, Petya Osenova, Paulo Rocha, Bogdan Sacaleanu, and Richard F. E. Sutcliffe. Overview of the CLEF 2007 Multilingual Question Answering Track. In Carol Peters, Valentin Jijkoun, Thomas Mandl, Henning Müller, Douglas W. Oard, Anselmo Peñas, Vivien Petras, and Diana Santos, editors, *CLEF*, volume 5152 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2007. ISBN 978-3-540-85759-4.

[GGM92] Terry Gaasterland, Parke Godfrey, and Jack Minker. Relaxation as a Platform for Cooperative Answering. *Journal of Intelligent Information Systems*, 1(3/4):293–321, 1992.

[Gnö00] Carsten Gnörlich. MultiNet/WR: A Knowledge Engineering Toolkit for Natural Language Information. Technical Report 278, University Hagen, Hagen, Germany, 2000.

[GWCL61] Bert F. Green, Jr., Alice K. Wolf, Carol Chomsky, and Kenneth Laughery. Baseball, an Automatic Question-Answerer. *International Workshop on Managing Requirements Knowledge*, 0:219, 1961.

[Har95] Donna Harman. The TREC Conferences. In *HIM*, pages 9–28, 1995.

[Har03] Sven Hartrumpf. *Hybrid Disambiguation in Natural Language Analysis*. Der Andere Verlag, Osnabrück, Germany, 2003. ISBN 3-89959-080-5.

[Har04] Sven Hartrumpf. Question Answering Using Sentence Parsing and Semantic Network Matching. In Carol Peters, Paul Clough, Julio Gonzalo, Gareth J. F. Jones, Michael Kluck, and Bernardo Magnini, editors, *CLEF*, volume 3491 of *Lecture Notes in Computer Science*, pages 512–521. Springer, 2004. ISBN 3-540-27420-0.

[Hel06] Hermann Helbig. *Knowledge Representation and the Semantics of Natural Language*. Cognitive Technologies. Springer, 2006. ISBN 978-3-540-24461-5.

[HF97] Birgit Hamp and Helmut Feldweg. GermaNet – a Lexical-Semantic Net for German. In *In Proceedings of ACL workshop Automatic Information Extraction and Building of Lexical Semantic Resources for NLP Applications*, pages 9–15, Somerset, NJ, USA, 1997. Association for Computational Linguistics.

[HG01] Lynette Hirschman and Robert Gaizauskas. Natural Language Question Answering: The View from Here. *Journal of Natural Language Engineering*, 7(4):275–300, 2001.

[HHO03] Sven Hartrumpf, Hermann Helbig, and Rainer Osswald. The Semantically Based Computer Lexicon HaGenLex - Structure and Technological Environment. *Traitement Automatique des Langues*, 44(2):81–105, 2003.

[Hil03] Thomas Hillenbrand. Citius altius fortius: Lessons learned from the Theorem Prover Waldmeister. In *Proceedings of the 4th International Workshop on First-Order Theorem Proving, number 86.1 in Electronic Notes in Theoretical Computer Science*, 2003.

[HMM+00] Sanda Harabagiu, Dan Moldovan, Rada Mihalcea, Mihai Surdeanu, and Vasile Rus. Falcon: Boosting Knowledge for Answer Engines. In *Proceedings of the 9th Text Retrieval Conference (TREC-9)*, pages 479–488, 2000.

[Hoi10] Dale Hoiberg, editor. *Encyclopædia Britannica*. Encyclopædia Britannica, Inc., 2010.

[HS00] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT 2000*, pages 283–292. IOS Press, 2000.

[Hur03] Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[HV11] Kryštof Hoder and Andrei Voronkov. Sine Qua Non for Large Theory Reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2011. ISBN 978-3-642-22437-9.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321154916.

[ISC+09] Radu Ion, Dan Stefanescu, Alexandru Ceausu, Dan Tufis, Elena Irimia, and Verginica Barbu Mititelu. A Trainable Multi-factored QA System. In Carol Peters, Giorgio Maria Di Nunzio, Mikko Kurimo, Djamel Mostefa, Anselmo Peñas, and Giovanna Roda, editors, *CLEF (1)*, volume 6241 of *Lecture Notes in Computer Science*, pages 257–264. Springer, 2009. ISBN 978-3-642-15753-0.

[KK98] Claude Kirchner and Hélène Kirchner, editors. *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Computer Science*. Springer, 1998. ISBN 3-540-64675-2.

[KN86] Deepak Kapur and Paliath Narendran. NP-Completeness of the Set Unification and Matching Problems. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495. Springer, 1986. ISBN 3-540-16780-3.

[Kor08] Konstantin Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 292–298, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-71069-1.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978. ISBN 0-13-110163-3.

[KS10] Konstantin Korovin and Christoph Sticksel. iProver-Eq: An Instantiation-Based Theorem Prover with Equality. In *5th International Joint Conference on Automated Reasoning, IJCAR 2010*, Lecture Notes in Computer Science, pages 196–202, Berlin / Heidelberg, 2010. Springer. ISBN 978-3-642-14202-4.

[Len95] Douglas B. Lenat. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[LMG94] Reinhold Letz, Klaus Mayr, and Christoph Goller. Controlled Integrations of the Cut Rule into Connection Tableaux Calculi. *Journal of Automated Reasoning*, 13(3):297–337, 1994.

[MB88] Rainer Manthey and François Bry. SATCHMO: A Theorem Prover Implemented in Prolog. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 415–434, London, UK, 1988. Springer-Verlag. ISBN 3-540-19343-X.

[McC92] William McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[McC97] William McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19:263–276, 1997.

[McC03] William McCune. *OTTER 3.3 Reference Manual*. Argonne National Laboratory, Argonne, Illinois, 2003.

[MCHM03] Dan I. Moldovan, Christine Clark, Sanda M. Harabagiu, and Steven J. Maiorano. COGEX: A Logic Prover for Question Answering. In *HLT-NAACL*, 2003.

[MGdPSPB+09] Ángel Martínez-González, César de Pablo-Sánchez, Concepción Polo-Bayo, María Vicente-Díez, Paloma Martínez-Fernández, and José Martínez-Fernández. The MIRACLE Team at the CLEF 2008 Multilingual Question Answering Track. In Carol Peters, Thomas Deselaers, Nicola Ferro, Julio Gonzalo, Gareth J. F. Jones, Mikko Kurimo, Thomas Mandl, Anselmo Peñas, and Vivien Petras, editors, *Evaluating Systems for Multilingual and Multimodal Information Access* , volume 5706, chapter 48, pages 409–420. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04446-5.

[MIL+97] Max Moser, Ortrun Ibens, Reinhold Letz, Joachim Steinbach, Christoph Goller, Johann Schumann, and Klaus Mayr. SETHEO and E-SETHEO - The CADE-13 Systems. *Journal of Automated Reasoning*, 18:237–246, 1997.

[Nie99] Robert Nieuwenhuis. Invited talk: Rewrite-based deduction and symbolic constraints. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 302–313. Springer, 1999. ISBN 3-540-66222-7.

[Nie05] Robert Nieuwenhuis, editor. *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Computer Science*. Springer, 2005. ISBN 3-540-28005-7.

[NK09] Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 67–72. Springer, 2009. ISBN 978-3-642-03358-2.

[NP01] Ian Niles and Adam Pease. Towards a Standard Upper Ontology. In *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2001.

[NR95] Robert Nieuwenhuis and Albert Rubio. Theorem Proving with Ordering and Equality Constrained Clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995.

[NRW98] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On Generating Small Clause Normal Forms. In Kirchner and Kirchner [KK98], pages 397–411. ISBN 3-540-64675-2.

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing Small Clause Normal Forms. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 335–367. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

[OB03] Jens Otten and Wolfgang Bibel. leanCoP: Lean Connection-based Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.

[Pas01] Dominique Pastre. MUSCADET 2.3: A Knowledge-Based Theorem Prover Based on Natural Deduction. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 685–689. Springer, 2001. ISBN 3-540-42254-4.

[Pau89] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[PFR+10] Anselmo Peñas, Pamela Forner, Álvaro Rodrigo, Richard F. E. Sutcliffe, Corina Forascu, and Cristina Mota. Overview of ResPubliQA 2010: Question Answering Evaluation over European Legislation. In Martin Braschler, Donna Harman, and Emanuele Pianta, editors, *CLEF (Notebook Papers/LABs/Workshops)*, 2010. ISBN 978-88-904810-0-0.

[PFS+09] Anselmo Peñas, Pamela Forner, Richard Sutcliffe, Álvaro Rodrigo, Corina Forăscu, Iñaki Alegria, Danilo Giampiccolo, Nicolas Moreau, and Petya Osenova. Overview of ResPubliQA 2009: Question Answering Evaluation over European Legislation. In *Proceedings of the 10th Cross-Language Evaluation Forum Conference on Multilingual Information Access Evaluation*, CLEF'09, pages 174–196, Berlin, Heidelberg, 2009. Springer. ISBN 3-642-15753-X, 978-3-642-15753-0.

[PHF+11] Anselmo Peñas, Eduard H. Hovy, Pamela Forner, Álvaro Rodrigo, Richard F. E. Sutcliffe, Corina Forascu, and Caroline Sporleder. Overview of QA4MRE at CLEF 2011: Question Answering for Machine Reading Evaluation. In Vivien Petras, Pamela Forner, and Paul D. Clough, editors, *CLEF (Notebook Papers/Labs/Workshop)*, 2011. ISBN 978-88-904810-1-7.

[PSS02] Francis Jeffry Pelletier, Geoff Sutcliffe, and Christian Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[PY03] David A. Plaisted and Adnan H. Yahya. A Relevance Restriction Strategy for Automated Deduction. *Artificial Intelligence*, 144(1-2):59–93, 2003.

[Ray94] W. Boyd Rayward. Visions of Xanadu: Paul Otlet (1868-1944) and Hypertext. *Journal of the American Society for Information Science*, 45(4):235–250, May 1994.

[Rob65a] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rob65b] John Alan Robinson. Automatic Deduction with Hyper-Resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[RV02] Alexandre Riazanov and Andrei Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[RV03] Alexandre Riazanov and Andrei Voronkov. Efficient Instance Retrieval with Standard and Relational Path Indexing. In Baader [Baa03], pages 380–396. ISBN 3-540-40559-3.

[Sch95] Uwe Schöning. *Logik für Informatiker*. Reihe Informatik. Spektrum Akademischer Verlag, 1995. ISBN 978-3-86025-684-8.

[Sch02] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.

[Sch04] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Proceedings of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving*. Elsevier Science, 2004.

[Sch09] Rolf Schwitter. Anaphora Resolution Involving Interactive Knowledge Acquisition. In Norbert E. Fuchs, editor, *CNL*, volume 5972 of *Lecture Notes in Computer Science*, pages 36–55. Springer, 2009. ISBN 978-3-642-14417-2.

[SCZ11] Mihai Surdeanu, Massimiliano Ciaramita, and Hugo Zaragoza. Learning to Rank Answers to Non-Factoid Questions from Web Collections. *Computational Linguistics*, 37(2):351–383, 2011.

[SK05] Alex Sinner and Thomas Kleemann. KRHyper - In Your Pocket. In Nieuwenhuis [Nie05], pages 452–457. ISBN 3-540-28005-7.

[SKW08] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A Large Ontology from Wikipedia and WordNet. *Elsevier Journal of Web Semantics*, 2008.

[SQ07] José Saias and Paulo Quaresma. The Senso Question Answering Approach to Portuguese QA@CLEF-2007. In *Proceedings of CLEF - Cross Language Evaluation Forum*, Budapest, Hungary, September 2007. ISSN: 1818-8044, ISBN: 2-912335-32-9.

[SS98] Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[SS06] Geoff Sutcliffe and Christian Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[SST+10] Geoff Sutcliffe, Martin Suda, Alexandra Teyssandier, Nelson Dellis, and Gerard de Melo. Progress Towards Effective Automated Reasoning with World Knowledge. In Hans W. Guesgen and R. Charles Murray, editors, *FLAIRS Conference*. AAAI Press, 2010.

[SSW+09] Martin Suda, Geoff Sutcliffe, Patrick Wischnewski, Manuel Lamotte-Schubert, and Gerard de Melo. External Sources of Axioms in Automated Theorem Proving. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *KI*, volume 5803 of *Lecture Notes in Computer Science*, pages 281–288. Springer, 2009. ISBN 978-3-642-04616-2.

[Sti73] Rona B. Stillman. The Concept of Weak Substitution in Theorem-Proving. *Journal of the ACM*, 20(4):648–667, October 1973.

[Sti87] Mark E. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4:353–380, 1987.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986. ISBN 0-201-12078-X.

[Sut08] Geoff Sutcliffe. The CADE-21 Automated Theorem Proving System Competition. *AI Communications*, 21(1):71–81, 2008.

[Sut09] Geoff Sutcliffe. The 4th IJCAR Automated Theorem Proving System Competition - CASC-J4. *AI Communications*, 22(1): 59–72, 2009.

[Sut10] Geoff Sutcliffe. The CADE-22 Automated Theorem Proving System Competition - CASC-22. *AI Communications*, 23(1): 47–59, 2010.

[Sut11] Geoff Sutcliffe. The 5th IJCAR automated theorem proving system competition - CASC-J5. *AI Communications*, 24(1): 75–89, 2011.

[Sut12] Geoff Sutcliffe. The CADE-23 Automated Theorem Proving System Competition - CASC-23. *AI Communications*, 25(1): 49–63, 2012.

[SWC00] Mark E. Stickel, Richard J. Waldinger, and Vinay K. Chaudhri. A Guide to SNARK, 2000.

[Tam98] Tanel Tammet. Towards Efficient Subsumption. In Kirchner and Kirchner [KK98], pages 427–441. ISBN 3-540-64675-2.

[TM10] Xavier Tannier and Véronique Moriceau. FIDJI @ ResPubliQA 2010. In *CLEF (Notebook Papers/LABs/Workshops)*, 2010.

[TP10] William Tunstall-Pedoe. True Knowledge: Open-Domain Question Answering Using Structured Knowledge and Inference. *AI Magazine*, 31(3):80–92, 2010.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988. ISBN 0-7167-8158-1.

[Voo01] Ellen M. Voorhees. Overview of TREC 2002. In *In Proceedings of the 11th Text Retrieval Conference (TREC 2002), NIST Special Publication 500-251*, pages 1–15, 2001.

[Wal] Russell Wallace. `http://code.google.com/p/ayane`.

[Wer03] Christoph Wernhard. System Description: KRHyper. Technical report, Universität Koblenz-Landau, 2003.

[Win71] Terry Winograd. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. Technical report, MIT, February 1971.

[WM10] Sarah Winkler and Aart Middeldorp. Termination Tools in Ordered Completion. In *5th International Joint Conference on Automated Reasoning, IJCAR 2010*, Lecture Notes in Computer Science, pages 518–532. Springer, 2010.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[WS08] Richard Waldinger and Jeff Shrager. Answering Science Questions: Deduction with Answer Extraction and Procedural Attachment. *AAAI Spring Symposium: Semantic Scientific Knowledge Integration*, 2008.

[WSH+07] Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Rostislav Rusev, and Dalibor Topic. System Description: SPASS Version 3.0. In *Automated Deduction - CADE-21, Proceedings*, pages 514–520, 2007.

Index