Learning Search Control Knowledge for Equational Deduction

Stephan Schulz

Institut für Informatik
der Technischen Universität München
Lehrstuhl für Informatik VIII

Learning Search Control Knowledge for Equational Deduction

Stephan Schulz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. T. Nipkow, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. E. Jessen
2.	Univ.-Prof. Dr. J. Avenhaus
	Universität Kaiserslautern

Die Dissertation wurde am 11.11.1999 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 9.2.2000 angenommen.

Preface

Modern computer systems are very good at following exact rules correctly, without tiring, and at amazing speed. However, there are problems for which this ability alone is not sufficient. As an example, consider strategy games such as chess or checkers. While these games are fully determined by the rules, novice players, even if they know these rules perfectly, cannot expect to play an even adequate game. Only with the experience gained from playing and studying different games does a player's performance increase. The same effect is visible with computer programs. Good chess programs do not rely on brute force alone. Instead, the have large libraries of openings and endgames, and use complex heuristic evaluation functions to guide the search for promising moves during midgame. These libraries and heuristics encode the experience of chess players as well as those of the programs' developers.

Something very similar can be observed in the field of mathematics. Students learn definitions and theorems. However, to apply this knowledge usefully, they also need to study examples of mathematical reasoning, and need to practice this kind of reasoning themselves. In fact, the non-formalized knowledge gained in this way is much more important than knowledge of the laws of any single mathematical structure. The equivalent to a chess program in mathematics is an *automated theorem prover*. Theorem provers try to show that a given formula is a logical consequence of a set of axioms. They are being applied not only to purely mathematical problems, but also to more practical domains like verification or linguistic analysis.

For most interesting logics, theorem provers have to search for a proof in an infinite search space. This search is typically guided by a heuristic evaluation function that selects the most promising of the different alternatives at every search state. The performance of a prover critically dependends on the suitability of this guiding heuristics. Most existing theorem provers implement a variety of different search heuristics. However, the selection of a suitable heuristics, and even more the creation of new heuristics for a given domain, is highly non-trivial, and typically requires significant work by an expert user. This is one of the reasons why the practical applicability of theorem provers has been limited in the past.

This thesis develops an approach to automate the task of creating suitable search heuristics for different domains by learning from previous proof experiences. It covers all aspects, from the generation of proof experiences and the suitable representation of search control knowledge to the selection of suitable experiences and the learning of evaluations for search decisions from these experiences. The implementation and evaluation of this approach show the significant improvements that can be achieved in this way.

While the current work is primarily aimed at theorem proving in clausal logic with equality, large parts of it can be similarly applied to other theorem provers and symbolic reasoning systems, and some of the results are interesting for the general machine learning community as well.

Munich, February 2000 Prof. Dr. Eike Jessen

Acknowledgments

First and foremost I have to thank Professor Dr. Eike Jessen, who acted as my primary advisor. He also enabled me to work in his research group and to develop my own ideas to an unusually large degree. I am also very grateful to Professor Dr. Jürgen Avenhaus, who volunteered to serve as the second advisor and who provided valuable input.

Particular thanks go to Ortrun Ibens and Jörg Denzinger, who have read parts of the thesis for both scientific content and grammatical and orthographic problems, and pointed out lots of minor errors and potential improvements. Similarly, Persefoni Tsetini has read a large part of the text for English language problems.

Among my co-workers, the discussions with Christoph Goller on different approaches to learning and the collaboration with Joachim Steinbach on (for me) tedious parts of the development and implementation of the equational theorem prover E have been particularly important.

This thesis would have looked very different without the stimulating and pleasant work environment provided by the other current and former members of our research group: Joachim Dräger, Bertram Fronhöfer, Marc Fuchs, Michael Greiner, Reinhold Letz, Max Moser, Manfred Schramm, Johann Schumann, Gernot Stenz, and Andreas Wolf

Last but not least, I must thank my friends and family, who have supported me in many different ways, and have tolerated my often incomprehensible babbling and my unconventional work schedule.

Abstract

Automated theorem provers for first order logic are increasingly being used in formal mathematics or for verification tasks. For these applications, efficient treatment of the *equality* relation is particularly important. Due to the special properties of the equality relation, the proof search is particularly hard for problems containing equality, even if state of the art calculi are used.

In this thesis we develop techniques to automatically learn good search heuristics to control the proof search of a superposition-based theorem prover for clausal logic with equality. We describe a variant of the superposition calculus and an efficient proof procedure implementing this calculus. An analysis of the choice points of this algorithm shows that the order in which new logical consequences are being processed by the prover is the single most important decision during the proof search.

We develop methods to extract information about important good and bad search decisions from existing proof searches. These search decisions are represented by signature-independed annotated *representative clause patterns*, which represent all analogous search decision by a single unique term. Annotations carry information about the role of the seach decisions in different proof attempts.

To utilize the stored knowledge for a new proof attempt, experiences generated from proof problems similar to the one at hand are extracted from the stored knowledge. The selected proof experiences, represented by a set of patterns with associated evaluations, are used as input for a new, hybrid learning algorithm which generates a *term space map*, a structure that allows the evaluation of new potential search decisions.

We experimentally demonstrate the performance of different variants of the term space mapping algorithm for artificial term classification problems as well as a siginificant gain in performance for the learning theorem prover E/TSM compared to the variant using only conventional search heuristics.

Contents

List of Tables

List of Figures

Table of Frequently Used Symbols and Notations

$\equiv$	Syntactic identity of two objects.
$\simeq$	Equality predicate symbol, usually used in infix notation.
$\not\simeq$	Shorthand for negated equality predicate symbol, $s \not\simeq t = \neg \simeq (s,t)$.
$\dot{\simeq}$	Shorthand for either the negated or the normal equality predicate symbol.
λ	The empty word or sequence.
$\mathcal{D}_{SP}$	The set of all **SP**-derivations (see Definition 2.35 on page 24.
$f(M)$	Image of a (multi-)set M under f, $f(M) = \{f(x) \mid x \in M\}$.
$id : M \to M$	The identity function, $id(x) = x$ for all $x \in M$.
$\|M\|$	Cardinality of a (multi-)set M.
$max_>(x,y)$	Maximum of x and y with respect to $>$. We omit $>$ if the context implies the ordering.
$max_>(M)$	Maximum of the elements of a (multi-)set (with respect to $>$).
$min_>(x,y)$	Minimum of x and y (with respect to $>$).
$min_>(M)$	Mimimum of the elements of a (multi-)set M (with respect to $>$).
$Mult(M)$	Set of multi-sets over M (Definition 2.5 on page 8).
$\mathbf{N}$	Set of natural numbers, $\mathbf{N} = \{0, 1, 2, \ldots\}$
$\mathbf{N}^+$	Set of natural numbers greater than 0.
$\mathbf{N}^\infty$	$\mathbf{N} \cup \{\infty\}$, $\infty > i$ for all $i \in \mathbf{N}$.
$O(t)$	Set of positions in a term t, Definition 2.14.
$\mathcal{P}_G$	Set of pathes in a graph G (Definition 2.9, page 11).
$\mathbf{R}$	Set of real numbers.
$\mathbf{R}^+$	Set of real numbers greater than or equal to 0.
$[a;b]$	A closed interval of real numbers, $[a;b] = \{x \in \mathbf{R} \mid a \leq x \leq b\}$.
2^M	Power set of a set M, i.e. the set of all subsets of M.

Chapter 1

Introduction

Automated theorem proving (ATP) systems try to answer the question about the validity of a given *hypothesis* under a set of *axioms*. For the most common case, both axioms and hypothesis are expressed as formulae of (a subset of) first-order predicate logic, and the answer to the question of semantic validity is searched for by syntactic manipulation of these formulae.

The last few years have seen a steady increase in the use of automated theorem provers in research and development. Theorem provers like Otter [McC94, MW97a], DISCOUNT [ADF95, DKS97], SPASS [WGR96, WAB+99] and SETHEO [LSBB92, MIL+97] are even beginning to make inroads into industrial use. They are being used for the verification of protocols [Sch97] and the retrieval of mathematical theorems [DW97] or software components [FS97] from libraries. Theorem provers are used to synthetize larger programs from standard building blocks and to prove the correctness of the resulting program systems [SWL+94, LPP+94, BRLP98]. This development is reflected in the creation of integrated interactive systems incorporating one or more automated theorem provers, a user interface with facilities for theory and subproof management, and often proof verification and representation components. Examples are the KIV system [Rei92, RSS95, Rei95], primarily for verification tasks or ILF [DGHW97] and Ωmega [BCF+97], which have primarily been developed for the support of mathematicians.

The most visible success of automatic theorem provers today is the celebrated proof of the Robbins algebra problem by EQP [McC97]. Successes like this demonstrate the power of current theorem proving technology.

However, despite the fact that ATP systems are able to perform basic operations at an enormous rate and can solve most simple problems much faster than any human expert, they still fail on many tasks routinely solved by mathematicians. Moreover, many of the more impressive successes require an experienced human user who selects a suitable prover configuration, often by trial and error.

The main reason for this is that a theorem prover has to *search* for a proof in a usually infinite *search space* with a very high branching factor, i.e. a very high number of possible choices at each choice point. Much previous work in theorem proving has been targeted at the development of refined calculi that restrict the number of possible inferences. However,

the semi-decidability of the underlying problem for most interesting logics restricts the potential for this approach, and even the most refined calculi typically are highly non-deterministic.

Most current theorem provers therefore use a small set of highly parameterized *heuristic evaluation functions* to guide the proof search. The selection of a proper function and set of parameters for a given problem (or problem domain) is based on experience of the human user, often supported by large and tedious sets of experiments.

The aim of this thesis is the development of techniques that on the one hand allow the automatic adaption of the search control component of a theorem prover to a given problem domain and on the other hand improve this component to increase the overall performance of the proof system.

To reach this goal we develop machine learning techniques to learn heuristic evaluation functions by extracting search control knowledge from examples of proof searches and apply these techniques in an equational proof system for full clausal logic.

1.1 Equational Theorem Proving

A particularly important problem in automated theorem proving is the handling of the *equality* relation. This relation plays an important role in modeling most mathematical or verification problems. Equality occurs e.g. in 1942 of the 3275 clausal problems in version 2.1.0 of the TPTP reference library of theorem prover problems [SSY94, SS97b], despite the fact that the TPTP is biased against equality encodings by the inclusion of large repositories of older problems that avoid equality. Moreover, functional programming languages like *Haskell* [Bir98, JHA+99, Tho99] are based on equational transformations. A program in such a language can be seen as a specification of a particular equality relation, and evaluation corresponds to the computation of a *normal form* with respect to this specification[1]. Naturally, verification of programs in such languages requires efficient handling of equality.

As the equality relation is a congruence relation, it is particularly hard to control. Symmetry and transitivity of the relation immediately allow infinite derivations. If the equality relation is modeled explicitly by including the necessary axioms, these axioms can typically be applied extremely often during a proof search and thus lead to a very early explosion in the search space.

Efficient handling of equality therefore requires special inference rules that directly implement some features of the relation. The most important of these rules is the *paramodulation rule* [RW69], which directly implements the replacing of terms by equal terms. Based on this principle, refined *superposition calculi* [BG90, BG94, NR92] have been developed. They combine techniques from conventional theorem proving, like *resolution* [Rob65] and *paramodulation*, with *ordering restrictions* and *rewriting* originally introduced in the context of *completion* [KB70, HR87, BDP89] for unit-equational theories.

[1] This relationship goes so far that the name of the Haskell-dialect *Gofer* is expanded as ***Go****od* ***f****or* ***e****quational* ***r****easoning*.

Superposition calculi are *saturating* calculi with strong contraction rules. The most elementary operation is the generation of new consequences from a set of axioms. The proof search terminates if either a proof has been found or the set of consequences is saturated, i.e. no relevant new facts can be added. *Contraction* allows the instant elimination of certain of the consequences that can be shown to be redundant for the proof search. The order of generating and contracting operations is only very weakly constrained by the calculus. A good *control* of these operations is critical for the success of a proof search.

1.2 Learning Search Control Knowledge

An exact definition of *learning* is difficult to give, as the term is used quite differently for humans and for computer programs. A human is said to have learned something if he or she, by observation or by being taught, is able to perform some action or reproduce some information he was previously unable to perform or produce. A definition of this broadness applied to a machine would cover both being programmed for a certain task (as a special case of being taught), and simple storage and retrieval of information as e.g. performed by any data base program or even word processor. Both of these tasks are trivial for modern computers, and require no serious reorganization or processing on behalf of the learner. We will therefore use a more strict definition: Learning is the process of acquiring knowledge by processing information and by structuring the accumulated data in a way that adequately represents the (relevant) concepts contained in this data (see [Sch95] for a discussion of other possible definitions).

The motivation for applying learning techniques to equational theorem proving is simple: Despite the strong restrictions on inferences in current superposition calculi, the branching factor and hence the difficulty of the search problem is usually much bigger in theorem proving with equality than in theorem proving without equality. Therefore, good control of the proof search process is even more critical in the case of equational theorem proving. However, finding good search control heuristics for theorem provers is a very difficult and time-consuming feat for human experts.

There is a variety of choices to be made both before and during the proof search for a superposition-based theorem prover. These include *term ordering*, *literal selection functions* and *rewriting strategy*. However, the most important of these choice points is the order in which generating inferences are performed. We attack this choice point by introduction of a *feedback cycle*, and by thus improving the evaluation of the potential search alternatives by using experiences from previous proof attempts.

As we are learning *heuristic control knowledge*, our method is orthogonal to any refinements at the calculus level, as e.g. techniques to further prune the search space with additional constraints or techniques introducing stronger inference rules. Due to the very small interface between learning component and inference engine, this approach is also compatible with any improvements in the implementation of the inference rules.

1.3 Conception of a Learning Theorem Prover

As stated above, we want to apply machine learning to improve the performance of a theorem prover. The most important choice point for standard saturating theorem proving procedures is the order in which new clauses are considered for generating inferences. The decisions at this choice point can be represented by individual clauses and are typically guided by a *heuristic evaluation function*. We want to improve the performance of a prover by learning good evaluation functions from experiences with multiple previous proof searches. In particular, we want to learn *proof search intrinsic* knowledge, i.e. knowledge gained from the analysis of the inference process, as opposed to *meta-knowledge* as e.g. the performance of a given strategy on a given proof problem. For this purpose we have to embed different methods and algorithms into a framework including proof analysis, proof generalization, proof example administration and selection, and knowledge application.

First, proof analysis techniques yield useful information about the importance, usefulness and cost of selecting different clauses in a given proof search. This allows us to assign a measure of expected usefulness for each clause, and to select a relatively small number of clauses for representing good and bad search decisions for a given proof search.

Secondly, we have developed *representative patterns* for clauses as a generalization to the term patterns introduced in [Sch95, DS96a, DS98]. Representative clause patterns allow us to abstract from irrelevant details of a clause and furthermore let us represent equivalent but syntactically different clauses by a unique pattern. Thus it becomes possible to efficiently apply knowledge about clauses in analogous situations in new proof attempts.

Thirdly, we use feature-based similarity criteria to extract a subset of all stored proof experiences from a potentially large knowledge base. This subset is then used by a new and fast term-based learning algorithm to construct a *term space map* that defines an evaluation function for clauses. This function is then used to modify a standard search heuristic to guide the proof search for new problems.

Many of the techniques we developed were first implemented prototypically in the DISCOUNT system. However, in order to have a stronger and more general proof system as a base, we have implemented the E equational theorem prover [Sch99b], a high performance equational theorem prover based on superposition and rewriting. In this thesis, we will only describe the generalized techniques tested in this new proof system and refer to pre-published reports for details of the older system.

1.4 Overview of the Thesis

After the short introduction and overview given in this chapter, Chapter 2 introduces the basic concepts for equational deduction. It serves both to establish our notation and to describe the refined superposition calculus that we have developed for use for the base inference engine of our prover. As we use fairly standard notation, readers familiar with equational theorem proving should be able to skip most of this chapter except the beginning of Section 2.7 (page 21), which details the calculus we based our system on.

The third chapter describes the basic learning cycle for theorem provers. We split the task of improving the performance of a theorem prover by learning into three different phases and discuss the different solutions to the problems in each phase. We also describe which choices we have made for our approach.

In the next chapter, we analyze the search problem for superposition-based theorem proving. We suggest a search algorithm based on the *given-clause* principle and analyze the choice points in this algorithm. As a result of this analysis, we identify the selection of the given clause (the next clause to process) as the single most important choice point as well as the choice point that is most likely to profit from learning *proof-intrinsic* knowledge. We also discuss existing, non-learning approaches for dealing with this choice point.

Chapter 5 describes ways to represent various kind of knowledge useful for search control. It introduces numerical features and one of the central concepts of this thesis, *clause patterns*. Clause patterns allow us to uniquely represent structurally similar clauses from different proof problems and over different signatures. The chapter also describes our way to represent search decision during a proof search as sets of annotated clauses.

The following chapter is the second central chapter. It introduces *learning by term space mapping*, a class of fast hybrid learning algorithms for terms. Term space mapping is based on partitioning the set of all terms into distinct classes and by extrapolating evaluations for terms in these classes from evaluations of terms in a training set.

Chapter 7 describes how we have integrated all elements into a superposition based prover that can learn good search control heuristics from its own experiences. The experimental results in the next chapter show that this new theorem prover significantly improves compared to the base system.

Finally, Chapter 9 discusses options for future work and the last chapter concludes the thesis.

Chapter 2

Basic Concepts of Equational Deduction

In this chapter, we will introduce the basic elements necessary to describe superposition-based theorem proving in clausal logic with equality.

Full first-order logic (see e.g. [CL73]) offers a rich language with complex, hierarchical formulae, a large set of operators, and the use of quantifiers. While this is desirable for the specification of problems, more uniform and efficient proof procedures can be developed for simpler languages. We therefore restrict our discussion to *clausal logic*, a subset of first-order predicate logic that eliminates quantifiers and allows only conjunctions of clauses (which are disjunctions of elementary literals) as formulae. Clausal logic is powerful enough to specify most proof problems directly, and various automatic procedures for the transformation of problem specifications from full first-order logic into clausal form exist [CL73, Boy92]. The fact that theorem provers applying such transformations have dominated the first-order category of the yearly CASC theorem prover competition [SS97a] since this category has been introduced in 1997 is ample evidence that this transformational approach is adequate.

In clausal theorem proving, the problem of showing that the axioms imply the hypothesis is usually reduced to show that a set of clauses (generated from the axioms and the negated and Skolemized goal) is unsatisfiable, i.e. that there is no possible interpretation which makes all clauses in the set true.

Well-known calculi for the proof search in clausal logic are e.g. *resolution* [Rob65] and *model elimination* [Lov78]. Superposition calculi have historically been developed by adding explicit *replacing of equals with equals* to resolution procedures while trying to control the search space explosion by adding constraints to restrict the number of possible or necessary inferences. Many of the techniques used have originally be developed in the context of term rewriting and completion for unit-equational theories, and have later been adapted to the general case.

2.1 General Preliminaries

In this section, we will introduce some basic concepts and results used throughout this work. In particular, we will cover binary relations, orderings and multi-sets.

Definition 2.1 (Binary relations)
A *binary relation* $\rightarrow$ over a set M is a subset of $(M \times M)$. We usually write $m \rightarrow n$ for $(m, n) \in \rightarrow$. Now assume a binary relation $\rightarrow$ over M:

- $\leftarrow \; = \{(n, m) | m \rightarrow n\}$ is the *inverse relation* of $\rightarrow$.
- $\leftrightarrow$ is the *symmetric closure* of $\rightarrow$, i.e. $\leftrightarrow \; = \leftarrow \cup \rightarrow$.
- $\overset{+}{\rightarrow}$ is the *transitive closure* of $\rightarrow$, and $\overset{*}{\rightarrow}$ is the *transitive and reflexive* closure of $\rightarrow$.
- Finally, $\overset{*}{\leftrightarrow}$ is the *transitive, reflexive and symmetric closure* of $\rightarrow$, i.e. the *equivalence relation* spanned by $\rightarrow$.
- Given two relations $\rightarrow_1$ and $\rightarrow_2$ over M, $\rightarrow_1 \circ \rightarrow_2 = \{(a, c) | \; \exists b \in M$ with $a \rightarrow_1 b, b \rightarrow_2 c\}$ denotes the *composition* of the two relations.
- $\rightarrow$ is called *terminating* (*Noetherian*, *well-founded*), if there exist no infinite sequence $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$.

◀

Orderings are a particular class of binary relations.

Definition 2.2 (Partial ordering, Quasi-ordering)
Let M be a set.

- A *partial ordering* $\geq$ over M is a reflexive, anti-symmetric and transitive binary relation over M. The *strict part of* $\geq$ is given by $> \; = \geq \setminus \{(x, x) | x \in M\}$.
- A *quasi-ordering* $\succsim$ over M is a reflexive and transitive binary relation over M. The *strict part* of $\succsim$ is given by $\succ \; = \succsim \setminus \precsim$, the equivalence part of $\succsim$ is given by $\approx \; = \succsim \cap \precsim$.

◀

Note that each partial ordering is a quasi-ordering and that the strict part of each quasi-ordering is a partial ordering.

Definition 2.3 (Total orderings)
Let $\geq_1$ and $\succsim_2$ be a partial ordering and a quasi-ordering over a set M, respectively.

- $>_1$ is called *total*, if $x >_1 y$ or $y >_1 x$ or $x = y$ for all $x, y \in M$.

- $\succsim_2$ is called *total up to* $\approx_2$, if $x >_2 y$ or $y >_2 x$ or $x \approx_2 y$ for all $x, y \in M$.

◀

There are many ways to extend orderings from a set of simple objects to composite objects. A particular class of such extended orderings are *lexicographic orderings.*

Definition 2.4 (Lexicographic orderings)
Let M be a set and let $\succsim$ be a quasi-ordering over M. We extend $\succsim$ to orderings $\succ_{lex}$ and $\succ_{llex}$ over M^*, i.e. over the set of finite tuples with elements from M. Let $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ be finite tuples over M.

- A is bigger than b in the *lexicographic extension* of $\succsim$, written as $A \succ_{lex} B$, if and only if there exist an $i \leq min(n, m)$ such that for all $j < i$ $a_j \approx b_j$ and either $i = m, i < n$ or $a_i \succ b_i$.

- Similarly, A is bigger than b in the *length-lexicographic extension* of $\succsim$, written as $A \succ_{llex} B$, if and only if $n > m$ or $n = m$ and $A \succ_{lex} B$.

◀

Sets are collections of unique objects. However, in theorem proving we often have to deal with situations where identical objects can occur multiple times. *Multi-sets* represent finite collections of arbitrary objects (which may occur more than once in a multi-set). Formally, multi-sets are defined as functions from a base set into the set **N** of natural numbers (with 0).

Definition 2.5 (Multi-sets)
Let M be a set.

- A *multi-set* over M is a function $A : M \to \mathbf{N}$ where $\{x | A(x) > 0\}$ is finite. We usually describe a multi-set by enumerating its elements in a set-like notation. We write $Mult(M)$ to denote the system of multi-sets over M.

- Most set operations are generalized to multi-sets. Let A, B be two multi-sets over M:
  - We write $x \in A$ if $A(x) > 0$.
  - The empty multi-set is written as $\emptyset$ or $\{\}$.
  - The cardinality of a multi-set A is $|A| = \sum_{x \in M} A(x)$.
  - $A \subseteq B$ if $A(x) \leq B(x)$ for all $x \in M$.
  - $(A \cup B)(x) = A(x) + B(x)$ for all $x \in M$.
  - $(A \cap B)(x) = min(A(x), B(x))$ for all $x \in M$.
  - $(A \backslash B)(x) = max(A(x) - B(x), 0)$ for all $x \in M$.

  - $(A \setminus\setminus B)(x) = \begin{cases} 0 & \text{if } B(x) > 0 \\ A(x) & \text{otherwise} \end{cases}$ for all $x \in M$.
  - $(f(A))(x) = \sum_{z \in \{y \in M | f(y) = x\}} A(z)$ for all $x \in M$. $f(A)$ is called the image of A under f.
- If A is a multi-set, $set(A) = \{x | A(x) > 0\}$ is the set of all elements of A.

◀

The definition of the image of a multi-set warrants an example:

Example: Let $A = \{0, 1, 1, 2, 2, 2\}$ be a multi-set over $\mathbf{N}$. We consider two functions, $f_1 : \mathbf{N} \to \mathbf{N}$ and $f_2 : \mathbf{N} \to \mathbf{N}$, defined by $f_1(x) = \lfloor \frac{x}{2} \rfloor$and $f_2(x) = 0$ for all x.

- $f_1(0) = 0$, $f_1(1) = 0$, $f_1(2) = 1$, hence $f_1(A) = \{0, 0, 0, 1, 1, 1\}$.
- $f_2(A) = \{0, 0, 0, 0, 0, 0\}$, and therefore
  - $(f_2(A))(0) = 6$
  - $(f_2(A))(x) = 0$ for all $x \in \mathbf{N}, x \neq 0$

If a set M underlying a multi-set is ordered, we can extend this ordering to the set of multi-sets over M [DM79]:

Definition 2.6 (Multi-set orderings)
Let M be a set. The relation $>>$ on $Mult(M)$ for a partial ordering $>$ on M is defined as follows. Assume $A, B \in Mult(M)$.

$A >> B$ if and only if there exist $X, Y \in Mult(M)$ with $X \neq \emptyset, X \subseteq A, B = (A \setminus X) \cup Y$, and for all $y \in Y$ exists an $x \in X$ with $x > y$.

◀

Example: Let $M = \{a, b, c, d\}$ be a set with $a > b > c > d$. Assume two multi-sets A and B, $A = \{a, b, c\}, B = \{b, b, b, b, b, c, c, d, d, d\}$. Then $A >> B$ because $B = (A \setminus \{a\}) \cup \{b, b, b, b, c, d, d, d\}$ and $a > b$, $a > c$, $a > d$.

As the authors of [DM79] show, multi-set orderings inherit several interesting properties from the base ordering:

Theorem 2.1 (Properties of $>>$)
Assume M, $>$, $>>$ as in Definition 2.6.

- $>>$ is a partial ordering.
- $>>$ is terminating if $>$ is terminating.
- $>>$ is total if $>$ is total.

2.2 Graphs and Trees

Most objects appearing in theorem proving, including terms, proofs, and even complete proof search protocols, can be represented as labeled *graphs* or *trees*. We also use graph and tree representations in learning algorithms, both for the representation of terms and for building hypotheses. Graphs consist of a set of *nodes* connected by *edges*.

Definition 2.7 (Graphs)

- Let K be an arbitrary set (of *nodes*) and let $E \subseteq (K \times K)$ a binary relation over K. Then the tuple $G = (K, E)$ is a (directed) *graph*. The elements of E are called *edges*.
- Let $G = (K, E)$ and $G' = (K', E')$ be two graphs. G' is a *subgraph* of G, if $K' \subseteq K$ and $E' \subseteq E$.
- Let $G = (K, E)$ be a graph, and let $a \in K$ be a node in G.
  - The set of *direct successors* of a is $succ(a) = \{b \in K | (a, b) \in E\}$.
  - The set of all successors of a is
$$succ^*(a) = succ(a) \cup \bigcup_{b \in succ(a)} succ^*(b)$$
  - Analogously, the set of *direct predecessors* of a is $pred(a) = \{b \in K | (b, a) \in E\}$.
  - Finally, the set of all predecessors of a is
$$pred^*(a) = pred(a) \cup \bigcup_{b \in pred(a)} pred^*(b)$$

◀

We usually use graphs to represent various objects from the proof process. For this purpose, it may be necessary to associate objects with graph nodes. Similarly, we will later represent the search space of a proof problem as a graph. In this case, different edges correspond to different search operations, and need different amount of effort to traverse. We model this by associating weights with the edges.

Definition 2.8 (Labeled and weighted graphs)

- Let L be a set (of *labels*) and let $G = (K, E)$ be a graph. A *labeled graph* is a tuple (G, l), where $l : K \to L$ is called a *label function*.
- Let $G = (K, E)$ be a graph, and let $w : E \to \mathbf{R}$ be a function assigning weights to the edges of G. Then (G, w) is a *weighted graph*.
- We treat an unweighted graph $G = (K, E)$ as a weighted graph (G, w_1) with $w_1 : E \to \mathbf{R}, w_1(e) = 1$ for all $e \in E$.

- If a graph G is both labeled and weighted, we write (G, l, w).

◀

Definition 2.9 (Paths, Distance)
Let $G = (K, E)$ be a graph.

- A finite *path* in G is a sequence $P = k_1, \ldots, k_n$ of nodes with

  1. $k_i \in K, 1 \leq i \leq n$, and
  2. $(k_i, k_i + 1) \in E, 1 \leq i \leq (n - 1)$

  We say P is a path from k_1 to k_n and denote the set of all paths from k_1 to k_n in G by $\mathcal{P}_G(k_1, k_n)$.

- If $P = k_0, \ldots, k_n \in \mathcal{P}_G$ is a path, we say that the nodes $k_0, \ldots, k_n$ are *on the path* P, and the edges $(k_0, k_1), \ldots, (k_{n-1}, k_n)$ are *parts of the path.*

- If $P = k_1, \ldots, k_n \in \mathcal{P}_G$ is a path and $(k_0, k_1) \in E$ is an edge, we also write $k_0.P = k_0, k_1, \ldots, k_n$ for the composite path from k_0 to k_n, i.e. we use '.' as a path constructor.

- Let (G, w) be a weighted graph. The *length* of a path, $len : \mathcal{P}_G \to \mathbf{R}$, is the sum of the weights of the edges connecting the elements of $P = k_0, \ldots, k_n$:

$$len(P) = \sum_{i=1}^{n-1} w((k_i, k_{i+1}))$$

  Note that for unweighted graphs the length of a path P is equal to the number of edges connecting the nodes in P.

- Let $((K, E), w)$ be a weighted graph. The *distance* of two nodes $a, b \in K$, $dist : K \times K \to \mathbf{R} \cup \{\infty\}$ is the length of the shortest path from a to b:

$$dist(a, b) = \begin{cases} min(\{len(P) | P \in \mathcal{P}_G(a, b))\} & \text{if } \mathcal{P}_G(a, b)) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

◀

We do not usually need arbitrary graphs, but can restrict ourselves to a class of graphs needed to model the theorem proving process. These graphs typically share a number of properties.

Definition 2.10 (Properties of graphs)
Let $G = (K, E)$ be a graph.

- G is *finite* if $|K| \in \mathbf{N}$.

- Let E^* be the reflexive, transitive and symmetric closure of E. G is called *connected* if $E^* = (K \times K)$.
- G is called *acyclic*, if there exists no non-trivial path from a node to itself, i.e. $P = a, \ldots, a$ implies $len(P) = 0$ for all P in $\mathcal{P}_G$.
- G is called *ordered*, if there is a total ordering on the direct successors of each node, i.e. the set of successors can be written as a sequence $succ(a) = k_1, k_2, \ldots$ for all $a \in K$. We usually denote the ordering by giving the sequence of successors.

◀

Definition 2.11 (Trees, Forests)

- A *tree* $T = (K, E)$ is a connected directed acyclic graph with
  - $pred(a) = \emptyset$ for an $a \in K$ (the *root* of T).
  - $|pred(a)| = 1$ for all $b \in K, b \neq a$.
- A *forest* is a (not necessarily connected) directed acyclic graph whose connected subgraphs are trees.

We can easily transform a forest $T = (K, E)$ into a tree $T' = (K \cup \{r\}, E \cup \{(r, a) | a \in K, pred(a) = \emptyset)\}$ with $r \notin K$. Therefore we will sometimes treat forests as trees without further remark. ◀

Trees inherit properties from graphs, i.e. if we speak of a finite or ordered tree, we mean a finite or ordered graph that is also a tree.

2.3 Terms

The most important building blocks of formulae are first-order *terms*. In a first-order specification, terms typically represent objects from the domain the we want to reason about. However, all structures handled by a typical automated theorem prover (literals, clauses, formulae) can easily and naturally be encoded as terms as well. Terms therefore are central for both the inference process and the learning algorithms introduced later.

Terms are build from a set of functions symbols, described by a *signature*, and a set of variables. In general, a signature can define terms with different *sorts*. However, we restrict our discussion to terms with a single sort only. For a more in-depth discussion of most of the topics of this and the following sections consult e.g. [BN98] and [Ave95], from which we borrow much of the notation.

Definition 2.12 (Signatures)

- A signature is a tuple $sig = (F, ar)$, where F is an enumerable set of *function symbols* (or *operators*) and $ar : F \to \mathbf{N}$ is a function describing the arity of the function symbols. Function symbols with the arity 0 are called *constants*. We write $sig = \{f_1/ar(f_1), \ldots, f_n/ar(f_n)\}$ as shorthand for $sig = (\{f_1, \ldots, f_n\}, ar)$.

- Two signatures $sig_1 = (F_1, ar_1)$ and $sig_2 = (F_2, ar_2)$ are *compatible* if $ar_1 \cup ar_2$ is a function, that is, if ar_1 and ar_2 agree on the function symbols occurring in both signatures.
- If $sig_1 = (F_1, ar_1)$ and $sig_2 = (F_2, ar_2)$ are compatible, we write $sig_1 \cup sig_2$ to denote the signature $(F_1 \cup F_2, ar_1 \cup ar_2)$. Similarly, $sig_1 \cap sig_2 = (F_1 \cap F_2, ar_1 \cap ar_2)$ and $sig_1 \backslash sig_2 = (F_1 \backslash F_2, ar_1 \backslash ar_2)$. We write $sig_1 \uplus sig_2$ to make clear that the two signatures being united do not share any function symbols.

◀

Definition 2.13 (Terms)
Let $sig = (F, ar)$ be a signature, and let V be an infinite enumerable set of *variable symbols* disjoint from F. ar is extended to a function $ar : F \cup V :\to \mathbf{N}$ by $ar(x) = 0$ for all $x \in V$.

- The set $Term(F)$ of *ground terms* over F is defined inductively: Let $f \in F$ be a function symbol with $ar(f) = n$ $(n \geq 0)$ and let $t_1, \ldots, t_n \in Term(F)$. Then $f(t_1, \ldots, t_n) \in Term(F)$.
- The set $Term(F, V)$ of *terms* over F and V is defined by $Term(F, V) = Term(F \cup V)$.
- Given a term t, $Var(t)$ is the set of variables occurring in t. We extend this to sets and multi-sets of terms in the obvious way, i.e. $Var(M) = \cup_{t \in M} Var(t)$ for a set or multi-set M.
- Given a term t, $Head(t)$ is the topmost symbol of t, i.e.
  - $Head(x) = x$ if $x \in V$
  - $Head(f(t_1, \ldots, t_n)) = f$ otherwise

In the following, we use the lower case letters x, y and z to denote variables. We use a, b, c, and d to denote constant symbols, and we omit braces from terms consisting of a single constant. Signatures are often given implicitly by the function symbols occurring in the terms. Unless otherwise mentioned, we will assume that F is a set of function symbols from some signature sig, and that V is a set of variables. We always demand that F contains at least one constant, so that $Term(F) \neq \emptyset$. ◀

With this definition, terms are recursive structures build from components that are themselves terms. We use *positions* (sequences of numbers) to describe these subterms.

Definition 2.14 (Positions, Subterms, Variable normalized)
Let $t \in Term(F, V)$ be a term and let λ be the empty sequence.

- The set $O(t)$ of positions in t is defined as follows:
  - If $t \equiv x \in V$, then $O(t) = \{\lambda\}$.
  - Otherwise $t \equiv f(t_1, \ldots, t_n)$. In this case $O(t) = \{\lambda\} \cup \{i.p | 1 \leq i \leq n, p \in O(t_i)\}$.

- For two positions $p, p' \in O(t)$, we say p' is *below* p if there exists a sequence q with $p' = pq$. In this case, we say p' is *strictly below* p, if $q \neq \lambda$.

- Now let $p \in O(t)$ be a position in t.

  - If $p = \lambda$ then $t|_p \equiv t$.
  - Otherwise $p \equiv i.q$ and $t \equiv f(t_1, \ldots, t_n)$. Then $t|_p \equiv t_i|_q$.

  We say $t' \in \mathit{Term}(F, V)$ is a *subterm* of t if $t' = t|_p$ for a $p \in O(t)$. We say t' is a *proper subterm* of t if $t' = t|_p$ for a $p \in O(t)$ with $p \neq \lambda$.

- Let $V = \{x_0, x_1, \ldots\}$ be a set of variables enumerated in the obvious way. A term t is called *variable normalized* if variables in t are picked in ascending order from V, i.e. if $\mathit{Var}(t) = \{x_0, \ldots, x_n\}$ and for all $i, j \in \{0, \ldots, n\}$ $p = \mathit{min}_{<_{lex}}\{q \mid t|_q = x_i\}, p' = \mathit{min}_{<_{lex}}\{q \mid t|_q = x_j\}, p <_{lex} p'$ implies $i < j$.

◀

One of the major operation in equational reasoning is the replacement of subterms by *rewriting* or *demodulation*, i.e. by the application of equations (compare Section 2.5).

Definition 2.15 (Term Replacing)
Let $s, t \in \mathit{Term}(F, V)$ and let $p \in O(t)$ be a position in t. $t[p \leftarrow s]$ is the term created by replacing $t|_p$ with s in t:

- $t[\lambda \leftarrow s] \equiv s$

- Otherwise $p \equiv i.q$ and $t \equiv f(t_1, \ldots, t_n)$. In this case $t[i.q \leftarrow s] = f(t_1, \ldots, f_i[q \leftarrow s], \ldots, f_n)$.

◀

As terms contain variables, another frequent operation is the instantiation of variables, i.e. the systematic substitution of variables with terms.

Definition 2.16 (Substitutions)

- A substitution is a function $\sigma : V \to \mathit{Term}(F, V)$ with the property that $\{x|\sigma(x) \neq x\}$ is finite. $\mathit{Dom}(\sigma) = \{x|\sigma(x) \neq x\}$ is called the *domain* of σ.

- A substitution σ is continued to a function $\sigma : \mathit{Term}(F, V) \to \mathit{Term}(F, V)$ by $\sigma(f(t_1, \ldots, f_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

- A substitution σ with $\sigma(\mathit{Dom}(\sigma)) \subseteq \mathit{Term}(F)$ is called a *ground substitution*. A substitution σ is called *grounding* for an expression t if $\sigma(t)$ does not contain any variables. We assume all ground substitution implicitly to be grounding for the expressions they are applied to.

- The composition of two substitutions σ and τ is given by $\sigma \circ \tau(x) = \sigma(\tau(x))$ for all $x \in V$.
- A substitution σ is *more general* than a substitution σ', if $\tau \circ \sigma = \sigma'$ for a substitution τ. σ is *strictly more general* than σ', if σ' is not more general than σ.
- A bijective substitution σ with $\sigma(x) = y, y \in V$ for all x is called a *variable renaming* for V. We denote the set of all variable renamings for a set of variables V with $\Sigma_{perm}(V)$.

◀

Applying substitutions to terms can make previously different terms equal. Depending on whether the substitution is applied to both terms or only one term, we speak of *matching* and *unification*.

Definition 2.17 (Match, Unifier)
Let $s, t \in Term(F, V)$ be two terms.

- A substitution σ with $\sigma(s) = t$ is called a *match* from s onto t. If such a substitution exists, we say s is *more general* than t, and t is called an *instance* of s. If s is more general than t, but not vice versa, we say s is *strictly more general* than t.
- A substitution σ with $\sigma(s) = \sigma(t)$ is called a *unifier* for s and t.
- If σ is a unifier for s and t and there is no unifier that is strictly more general than σ, we call σ a *most general unifier* for s and t. If a most general unifier for two terms exists, it is computable and unique modulo variable renamings [Rob65]. We therefore speak of *the* most general unifier for two terms and denote it by $mgu(s, t)$.

◀

Orderings on terms play an important role in equational theorem proving. They are used to restrict possible applications of equations and other inference steps, and to guarantee the completeness of proof procedures. Moreover, they are also necessary for the efficient implementations of many store and retrieval information for both inference machines and learning algorithms.

Three simple orderings on terms are given by the subterm relation, the relation ship between terms and their instantiation, and the combination of these two relations.

Definition 2.18 (Subterm and Subsumption orderings)
Let $s, t \in Term(F, V)$ be two terms. We define the following orderings on terms:

- The *subterm ordering* $\geq_{TT}$ is given by $s \geq_{TT} t$ if and only if there exists a $p \in O(s)$ with $s|_p = t$.
- The *subsumption ordering* $\geq_{SS}$ is given by $s \geq_{SS} t$ if and only if there exists a substitution σ with $s = \sigma(t)$.

◀

The subterm ordering is a partial ordering, subsumption defines a quasi-ordering.

A particularly important class of orderings for equational theorem proving is formed by so-called *reduction orderings*. Reduction orderings are compatible with the term structure and with instantiation. They can be used to restrict the application of equations during the proof process in a consistent way.

Definition 2.19 (Rewrite ordering, Reduction ordering)

- A *rewrite ordering* $>$ on $Term(F, V)$ is a partial ordering that is compatible with the term structure and stable under substitutions:
  1. $t > t'$ implies $s[p \leftarrow t] > s[p \leftarrow t']$ for all $s, t, t' \in Term(F, V), p \in O(s)$.
  2. $s > t$ implies $\sigma(s) > \sigma(t)$ for all $s, t \in Term(F, V)$ and all substitutions σ.
- A terminating rewrite ordering is called a *reduction ordering*.
- A *simplification ordering* is a reduction ordering $>$ that contains the subterm ordering, i.e. $>_{TT} \subseteq >$.
- A *ground reduction ordering* is a reduction ordering that is total on ground terms. Note that a ground reduction ordering is necessarily a simplification ordering on ground terms.

◀

2.4 Equations and rewrite systems

An equality relation allows the replacing of *equals with equals*. Equality relations over terms allow the replacing of subterms with equivalent ones. They are typically described by sets of *equations*.

Definition 2.20 (Equation, Congruence, *E*-equality)

- An *equation* is a pair of terms $(s, t) \in (Term(F, V) \times Term(F, V))$. We write an equation as $\simeq (s, t)$ or, more frequently, as $s \simeq t$ for the reserved predicate symbol $\simeq$, and implicitly consider equations to be symmetric, i.e. $s \simeq t$ and $t \simeq s$ are equivalent.
- A *negated equation* or *inequation* is a pair of terms (s, t) as well. We write a negated equation as $\not\simeq (s, t)$ or as $s \not\simeq t$. As with equations, we consider negated equations to be symmetric.
- A *congruence relation* $\sim$ on terms is an equivalence relation which is compatible with the term structure and substitutions:
  - $t \sim t'$ implies $s[p \leftarrow t] \sim s[p \leftarrow t']$ for all $s, t, t' \in Term(F, V), p \in O(s)$.

– $s \sim t$ implies $\sigma(s) \sim \sigma(t)$ for all $s, t \in \mathit{Term}(F, V)$ and all substitutions σ.

- A set of equations E over $\mathit{Term}(F, V)$ defines the relation $=_E$ (E-equality) as the smallest congruence that includes E.

◀

Equations are symmetrical and can be applied in two different directions. If we restrict this property by applying a reduction ordering, computation within the structure defined by a set of equations becomes much simpler.

Definition 2.21 (Rewrite relation)
Let $>$ be a reduction ordering and let E be a set of equations.

- A *rewrite relation* $\Longrightarrow$ is a binary relation on terms with the property that $s \Longrightarrow t$ implies $u[p \leftarrow \sigma(s)] \Longrightarrow u[p \leftarrow \sigma(t)]$.
- A rewrite relation $\Longrightarrow$ is *compatible with* $>$ if $\Longrightarrow \subseteq >$.
- E is called a rewrite system (with respect to $>$), if $s > t$ or $t > s$ for all $s \simeq t \in E$. In this case we also call the elements in E *rewrite rules*.
- E and $>$ define a rewrite relation $\Longrightarrow_E$: $s[p \leftarrow \sigma(l)] \Longrightarrow_E s[p \leftarrow \sigma(r)]$ iff $l \simeq r \in E$ and $\sigma(l) > \sigma(r)$.

◀

We distinguish between terms that can be rewritten with a certain rewrite relation and terms that cannot be modified:

Definition 2.22 (Reducible, Normal form)
Let $\Longrightarrow$ be a rewrite relation on $\mathit{Term}(F, V)$.

- A term t with $t \Longrightarrow s$ for some s is called *reducible (with respect to $\Longrightarrow$)*. A term that is not reducible is called *irreducible* or *in normal form*.
- If $s \overset{*}{\Longrightarrow} t$ and t is irreducible, then t is called a *normal form* of s.

◀

If we are dealing with a rewrite relation that is induced by a set of rules or equations and a reduction ordering, it is interesting to know exactly *where* in the term a rule or equation can be applied.

Definition 2.23 (Top-reducible)
Let $>$ be a reduction ordering and let E be a set of rewrite rules or equations.

- A term s is called *top-reducible at position p* (with respect to E and $>$), if there exists $(l, r) \in E$ such that $t|p = \sigma(l)$ and $\sigma(l) > \sigma(r)$.

- A term is called just *top-reducible* (with respect to E and $>$) if it is top-reducible at position λ. A term that is not top-reducible is called top-irreducible.

◀

E-equality is, in general, undecidable. However, it is decidable if we can compute unique normal forms for all terms.

Definition 2.24 (Confluence, Church-Rosser property)

- A relation $\Longrightarrow$ is called confluent, if all divergences can be joined again: $(\overset{*}{\Longleftarrow} \circ \overset{*}{\Longrightarrow}) \subseteq (\overset{*}{\Longrightarrow} \circ \overset{*}{\Longleftarrow})$.
- A relation $\Longrightarrow$ is called locally confluent, if all local (one-step) divergences can be joined: $(\Longleftarrow \circ \Longrightarrow) \subseteq (\overset{*}{\Longrightarrow} \circ \overset{*}{\Longleftarrow})$.
- A relation $\Longrightarrow$ has the *Church-Rosser property*, if $\overset{*}{\Longleftrightarrow} \subseteq (\overset{*}{\Longrightarrow} \circ \overset{*}{\Longleftarrow})$.

◀

The above properties are related, in fact, for terminating relations they are all equivalent:

Theorem 2.2 (Confluence properties)

- A relation is confluent if and only if it has the Church-Rosser property.
- A terminating relation is confluent if and only if it is locally confluent.

Definition 2.25 (Convergence)
A relation that is both terminating and Church-Rosser is called *convergent*.

◀

2.5 Clauses and Formulae

Terms are used to model domain objects and functions over them, with each term representing a class of objects and each ground term a single object. We now define *atoms* and *literals* to represent relations over objects. Literals are combined in *clauses* and allow us to state propositions over these relations, and (multi-)sets of clauses (*formulae*) finally correspond to specification and query of a proof problem. As we are interested in applying equational reasoning techniques, non-equational relations are encoded as equations.

Definition 2.26 (Atoms, Literals)
Let $S = F \uplus P$ be the union of two finite sets of symbols (*function symbols* and *predicate symbols*, respectively) with $F \cap P = \emptyset$ and a special symbol $\top \in P$. Let $sig = (S, ar)$ be a signature (with $ar(\top) = 0$) and let V be a set of variables disjoint from S.

- A (non-equational) *atom* (over *sig* and V) is a term $\mathcal{P}(t_1, \ldots, t_n) \in Term(S, V)$ with $\mathcal{P} \in P$ and $t_i \in Term(F, V), 1 \leq i \leq n$. The *equational representation* of a non-equational atom $\mathcal{A}$ is the equation $\mathcal{A} \simeq \top$.

- A *(equational) literal* $\mathcal{L}$ (over *sig* and V) is either an equation $s \simeq t$ (a *positive literal*) or a negated equation $s \not\simeq t$ (a *negative literal*) with $s, t \in Term(S, V)$. We will in practice restrict ourselves to literals where symbols from P only occur as head symbols of terms of the equational representation of a non-equational atom.
- We write $Literal(F, P, V)$ to describe the set of all literals for given sets of symbols F, P and V.

We will usually assume the set V of variables to be implicitly given.

◀

Clauses (disjunctions of literals) allow us to make conditional statements or specify alternatives for the relations represented by predicate symbols. Formally, clauses are *multi-sets* of literals.

Definition 2.27 (Clauses)
Assume V and *sig* with F and P as in Definition 2.26.

- A (equational) *clause* $\mathcal{C}$ over *sig* and V is a multi-set of literals. We implicitly assume $\mathcal{C}$ to be a disjunction of literals, and write $\mathcal{C} = \mathcal{L}_1 \vee \mathcal{L}_2 \vee \ldots \vee \mathcal{L}_n$ for $\mathcal{C} = \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n\}$. We write $\mathcal{C} = \mathcal{L} \vee \mathcal{C}'$ with $\mathcal{C}' = \mathcal{C} \backslash \mathcal{L}$ if $\mathcal{L} \in \mathcal{C}$. The set of all clauses over a signature is denoted by $Clause(F, P, V)$.
- The empty clause $\{\}$ is written as $\Box$.
- If $\mathcal{C}$ is a clause, we write $\mathcal{C}^+$ to denote the positive literals in $\mathcal{C}$ and $\mathcal{C}^-$ to denote the negative literals.
- A clause that contains only positive literals is called a *positive clause*.
- Similarly, a clause that contains only negative literals is a *negative clause*.
- A clause that contains at most one positive literal is called *Horn*.
- A clause that contains a single literal is called a *unit clause*. We will sometimes speak of a positive unit clause as a rewrite rule (if the two terms are comparable in some reduction ordering) or an equation, i.e. we will treat the clause as its single literal.
- If $\sigma(\mathcal{C}) = \mathcal{C}'$ for a variable renaming σ, we call $\mathcal{C}$) and $\mathcal{C}'$ *variants* (of each other). We will usually identify variants unless mentioned otherwise.

◀

While clauses represent individual propositions and hypotheses over a modeled structure, formulae describe a complete proof problem over these structure.

Definition 2.28 (Formulae)
Assume V and *sig* with F and P as before.

- A *formula* $\mathcal{F}$ (in clause normal form) over *sig* and V is a set of clauses. The clauses of a formula are implicitly considered to be conjunctively connected. We write $\mathcal{F} = \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ to represent the clause $\mathcal{F} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$.
- We say a formula is a *unit*-formula, if all clauses in the formula are unit.
- Similarly, as formula is called *Horn*, if all clauses in the formula are Horn.
- A formula is called a *general* formula, if it is not Horn, i.e. if it contains at least one non-Horn clause.

◀

Formulae in (refutational) automated theorem proving typically consist of two separate classes of clauses: Clauses describing the theory in which we want to reason (the *axioms*, forming the *specification* of an algebraic structure), and the *query* or *goal*, a set of clauses generated by negating the hypothesis. As formulae correspond to proof problems, we will sometimes use the terms interchangably, and speak e.g. of a *Horn problem* as shorthand for *a proof problem formalized as a Horn formula.*

2.6 Semantics

Formulae (in particular specifications) are used to describe algebraic structures, with each formula potentially describing an infinite number of structures. For theorem proving purposes, we are only interested in a particular class of these structures, with elements that are build from the symbols used in the specification.

Definition 2.29 (Interpretation, Model)
Assume a signature $sig = (S, ar)$ with at least one constant function symbol.

- A *(Herbrand equality) interpretation* is a congruence relation $\sim_I$ over $Term(S)$.
- An interpretation $\sim_I$ *satisfies* a ground clause $\mathcal{C}$ over *sig* if either $s \simeq t \in \sim_I$ for a positive literal $s \simeq t \in \mathcal{C}$ or $s \not\simeq t \notin \sim_I$ for a negative literal $s \not\simeq t \in \mathcal{C}$.
- An interpretation satisfies a non-ground clause $\mathcal{C}$ if it satisfies all ground instances of $\mathcal{C}$.
- An interpretation $\sim_I$ satisfies a formula $\mathcal{F}$ if it satisfies all clauses in $\mathcal{F}$. In this case, $\sim_I$ is called a *(Herbrand equality) model* of $\mathcal{F}$ and $\mathcal{F}$ is *satisfiable*.
- A formula that does not have a model is called *unsatisfiable*.

◀

This definitions immediately imply properties of the empty clause and the empty formula:

Corollary: The empty clause is not satisfied by any interpretation, the empty formula is satisfied by all interpretations.

The main mechanism in deduction is to infer new clauses from existing ones. In order for a new clause to be a logical consequence of an existing formula, it has to be satisfied by all structures that satisfy the original formula.

Definition 2.30 (Logical consequence)
Assume clauses $\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{C}$ over a signature *sig*. If each interpretation $\sim_I$ that satisfies $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ also satisfies $\mathcal{C}$, we say that $\mathcal{C}$ is a *logical consequence* of $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ and write $\mathcal{C}_1, \ldots, \mathcal{C}_n \models \mathcal{C}$. ◀

2.7 Superposition-Based Theorem Proving

We will now present the calculus **SP**. It is a variant of the *superposition calculi* decribed in [BG90, BG94]. Superposition calculi allows us to refute any unsatisfiable set of clauses by deriving the empty clause, i.e. by making the unsatisfiability obvious and explicit. In these calculi, a clause is essentially viewed as a set of conditional equations, were each positive literal in turn is seen as a potential rewrite rule and the remaining literals play the role of positive and negative conditions. The basic operation is the replacing of equals with equals, where term orderings are used to constrain this application of equality. Moreover, term orderings are extended to orderings on literals to determine in which order equations will be applied or conditions solved, and to orderings on clauses, which are used to introduce a strong approximation to the concept of *redundancy*.

Our version of the superposition calculus differs from the system $\mathcal{E}$ given in [BG94] only in some details. First, we sometimes use weaker restrictions on generating inferences, as our experiments with an actual implementation showed that the marginal improvements in search space reduction do not warrant the increased cost of checking the original stronger restrictions. Secondly, we have integrated explicit contraction rules into the calculus and in two cases allow stronger simplifications than suggested in [BG94]. Thirdly, we use an extended notion of *selection functions* and *eligible literals* that again allow redundant, but in practice often useful inferences.

We have implemented **SP** in the equational theorem prover E, the system we developed in the course of this work. Experimental results described in [Sch99b] as well as the performance of the prover in the CASC theorem prover competition show that the calculus is suitable for a high-performance theorem prover.

Definition 2.31 (Literal ordering, Clause ordering)
Let $>$ be a reduction ordering on $Term(F \uplus P, V)$.

- The *multi-set representation* of an equation $s \simeq t$ is $M(s \simeq t) = \{\{s\}, \{t\}\}$. Similarly, the multi-set representation of an inequation $s \not\simeq t$ is $M(s \not\simeq t) = \{\{s, t\}\}$. The multi-set representation of a clause $\mathcal{C} = \mathcal{L}_1 \vee \ldots \vee \mathcal{L}_n$ is $M(\mathcal{C}) = \{M(\mathcal{L}_1), \ldots, M(\mathcal{L}_2)\}$.

- We extended $>$ to an ordering $>_L$ on literals as follows: Assume two literals $\mathcal{L}_1, \mathcal{L}_2 \in \mathit{Literal}(F, P, V)$. $\mathcal{L}_1 >_L \mathcal{L}_2$ iff $M(\mathcal{L}_1) >> M(\mathcal{L}_2)$.
- Finally, we define $>_C$ on clauses by $\mathcal{C}_1 >_C \mathcal{C}_2$ iff $M(\mathcal{C}_1) >_L >_L M(\mathcal{C}_2)$ for $\mathcal{C}_1, \mathcal{C}_2 \in \mathit{Clause}(F, P, V)$.

◀

By construction, the clause ordering shares stability properties with the simplification ordering:

Theorem 2.3 (Properties of $>_L$ and $>_C$)
- $>_L$ is stable with respect to substitutions and total on ground literals.
- $>_C$ is stable with respect to substitutions and total on ground clauses.

As witten above, a clause can be seen as a set of conditional equations. Each of these equations can only contribute to the final proof if all its conditions are met. We can therefore restrict processing of a clause with at least one negative literal to trying to solve some of the negative literals. This restriction is described by means of a *selection function*, which maps a clause to a (multi-)subset of itself:

Definition 2.32 (Selection functions)
$\mathit{sel} : \mathit{Clauses}(F, P, V) \to \mathit{Clauses}(F, P, V)$ is a *selection function*, if it has the following properties for all clauses $\mathcal{C}$:

- $\mathit{sel}(\mathcal{C}) \subseteq \mathcal{C}$.
- If $\mathit{sel}(\mathcal{C}) \cap \mathcal{C}^- = \emptyset$, then $\mathit{sel}(\mathcal{C}) = \emptyset$.

We say that a literal $\mathcal{L}$ is *selected* (with respect to a given selection function) in a clause $\mathcal{C}$ if $\mathcal{L} \in \mathit{sel}(\mathcal{C})$. ◀

We will use two kinds of restrictions on deducing new clauses: One induced by ordering constraints and the other by selection functions. We combine these in the notion of *eligible literals*.

Definition 2.33 (Eligible literals)
Let $\mathcal{C} = \mathcal{L} \vee \mathcal{R}$ be a clause, let σ be a substitution and let *sel* be a selection function.

- We say $\sigma(\mathcal{L})$ is *eligible for resolution* if either
  - $\mathit{sel}(C) = \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(\mathcal{C})$ or
  - $\mathit{sel}(C) \neq \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(\mathit{sel}(\mathcal{C}) \cap \mathcal{C}^-)$ or
  - $\mathit{sel}(C) \neq \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(\mathit{sel}(\mathcal{C}) \cap \mathcal{C}^+)$.
- $\sigma(\mathcal{L})$ is *eligible for paramodulation* if $\mathcal{L}$ is positive, $\mathit{sel}(C) = \emptyset$ and $\sigma(\mathcal{L})$ is strictly $>_L$-maximal in $\sigma(\mathcal{C})$.

◀

We will describe the superposition calculus as a set of inference rules, describing transitions between (multi-)sets of clauses, and a fairness condition which ensures that the derivation process will eventually generate the empty clause from an unsatisfiable clause set. There are two distinct kinds of inference rules, *generating* rules and *contracting* rules. Generating rules allow us to deduce new clauses from existing ones. They are necessary to guarantee the completeness of the calculus, i.e. to ensure that we can find a proof if there exists one. Contracting rules eliminate or simplify existing clauses, thereby pruning the search space.

Definition 2.34 (Inference system)

- A generating inference rule is a deduction scheme of the form

  $$\frac{\texttt{<precondition>}}{\texttt{<conclusion>}} \quad \text{if } \texttt{<condition>},$$

  where `<precondition>` describes a set of clauses and `<conclusion>` a single clause. It allows us to *add* the clause from the conclusion to a clause set already containing clauses of the precondition if the condition is met.

- A contracting inference rule is a deduction scheme of the form

  $$\frac{\texttt{<precondition>}}{\texttt{<conclusions>}} \quad \text{if } \texttt{<condition>},$$

  where both `<precondition>` and `<conclusions>` describe sets of clauses. It allows us to *replace* the clauses of the precondition in a clause set with the clauses from the conclusion if the condition is met.

- We implicitly assume all clauses in the precondition of an inference rule to be variable disjoint. In practice, this can be easily achieved by variable renaming.

- An inference rule is *correct* if the clauses in the conclusion are logically implied by the clauses in the precondition.

- An inference system I is a set of inference rules. It is correct if all its rules are correct.

- An *inference (in I)* is an application of an inference rule. An *instance* of an inference with premises $\mathcal{C}_1, \ldots, \mathcal{C}_n$ and conclusion $\mathcal{C}'_1, \ldots, \mathcal{C}'_n$ is a corresponding inference with premises $\sigma(\mathcal{C}_1), \ldots, \sigma(\mathcal{C}_n)$ and conclusion $\sigma(\mathcal{C}'_1), \ldots, \sigma(\mathcal{C}'_n)$.

- If a set S of clauses can be transformed into a set S' by an inference in I, we write $S \vdash_I S'$. A (finite or countably infinite) sequence $S_0 \vdash_I S_1 \vdash_I \ldots$ is called an *I-derivation*.

◀

The following definition defines the deductions possible in the **SP** calculus:

Definition 2.35 (The inference system SP)
Let $>$ be a total simplification ordering (extended to orderings $>_L$ and $>_C$ on literals and clauses) and let *sel* be a selection function. The inference system **SP** consists of the following inference rules:

- *Equality Resolution*:

$$\text{(ER)}\ \frac{u \not\simeq v \vee R}{\sigma(R)}$$

if $\sigma = mgu(u, v)$ and $\sigma(u \not\simeq v)$ is eligible for resolution.

- *Superposition into negative literals*:

$$\text{(SN)}\ \frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

- *Superposition into positive literals*:

$$\text{(SP)}\ \frac{s \simeq t \vee S \quad u \simeq v \vee R}{\sigma(u[p \leftarrow t] \simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

- *Equality factoring*:

$$\text{(EF)}\ \frac{s \simeq t \vee u \simeq v \vee R}{\sigma(t \not\simeq v \vee u \simeq v \vee R)}$$

if $\sigma = mgu(s, u)$, $\sigma(s) \not> \sigma(t)$ and $\sigma(s \simeq t)$ eligible for paramodulation.

- *Rewriting of negative literals*:

$$\text{(RN)}\ \frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R}$$

if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$.

- *Rewriting of positive literals*[1]:

[1] A stronger version of (RP) is proven to maintain completeness for Unit and Horn problems and is generally believed to maintain completeness for the general case as well [Bac98]. However, the proof of completeness for the general case seems to be rather involved, as it requires a very different clause ordering than the one introduced in Definition 2.31, and we are not aware of any existing proof in the literature. The variant rule allows rewriting of maximal terms of maximal literals under certain circumstances:

$$\text{(RP')}\ \frac{s \simeq t \quad u \simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \simeq v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ and if $u \simeq v$ is not eligible for resolution or $u \not> v$ or $p \neq \lambda$ or σ is not a variable renaming.

This stronger rule is implemented successfully by both E and SPASS [Wei99].

(RP) $\dfrac{s \simeq t \quad u \simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \simeq v \vee R}$ if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$, and if $u \simeq v$ is not eligible for resolution or $u \not> v$ or $p \neq \lambda$.

- *Clause subsumption*:

(CS) $\dfrac{T \quad R \vee S}{T}$ if $\sigma(S) = T$ for a substitution σ or $\forall s \simeq t \in \sigma(S) : s \simeq t \in T$ for a substitution σ that is not a variable renaming.

- *Equality subsumption*:

(ES) $\dfrac{s \simeq t \quad u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t}$

- *Simplify-reflect*[2]:

(SR) $\dfrac{s \simeq t \quad u[p \leftarrow \sigma(s)] \not\simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t, R}$

- *Tautology deletion*:

(TD) $\dfrac{C}{}$ if C is a tautology[3].

- *Deletion of duplicate literals*:

(DD) $\dfrac{s \simeq t \vee s \simeq t \vee R}{s \simeq t \vee R}$

- *Deletion of resolved literals*:

(DR) $\dfrac{s \not\simeq s \vee R}{R}$

[2]In practice, this rule is only applied if $\sigma(s)$ and $\sigma(t)$ are $>$-incomparable – in all other cases this rule is subsumed by (RN) and the deletion of resolved literals (DR).

[3]This rule can only be implemented approximately, as the problem of recognizing tautologies is only semi-decidable in equational logic. The latest versions of E try to detect tautologies by checking if the ground-completed negative literals imply at least one of the positive literals, as suggested in [NN93].

We write $\mathbf{SP}(N)$ to denote the set of all clauses that can be generated with one generating inference from I on a set of clauses N, $\mathcal{D}_{SP}$ to denote the set of all **SP**-derivations, and $\mathcal{D}_{\overline{SP}}$ to denote the set of all finite **SP**-derivations.

◀

The inference system **SP** is easily shown to be correct, i.e. to add only logical consequences to a set of clauses:

Theorem 2.4 (Correctness of SP)
If $N \vdash_{\mathbf{SP}} N'$, then $N \models \mathcal{S}$ for all clauses $\mathcal{S} \in N'$.

Showing that **SP** is *refutationally complete*, i.e. that it is able to derive the empty clause from any unsatisfiable set of clauses, is more difficult. We will base our proof heavily on the one presented in [BG94] and only discuss some points where our calculus extends the one discussed in this paper.

The justification for contracting rules is that they only remove clauses that are *superfluous* or *redundant* in the sense that they are not necessary to describe the essential properties of a *saturated* system of clauses. It is not generally possible to identify all such clauses, however, the concept of *compositeness* gives us a very strong approximation of such redundancy.

Definition 2.36 (Composite clauses)
Let N be a set of clauses and let $\mathcal{C}$ be a ground clause.

- The clause $\mathcal{C}$ is called *composite with respect to* N, if there exist ground instances $\sigma_1(\mathcal{C}_1), \ldots, \sigma_n(\mathcal{C}_n)$ of clauses in N such that
  1. $\{\sigma_1(\mathcal{C}_1), \ldots, \sigma_n(\mathcal{C}_n)\} \models \mathcal{C}$ and
  2. $\mathcal{C} <_C \sigma_i(\mathcal{C}_i)$ for all $i \in \{1, \ldots, n\}$.
- A non-ground clause is called composite with respect to N if all its ground instances are.

◀

If a clause is composite with respect to the set of clauses at one state during the proof process, it keeps this property if we add new clauses or remove clauses which are themselves composite:

Theorem 2.5 (Stability of compositeness I)
Let N be a set of clauses and let $\mathcal{C}$ be a clause that is composite with respect to N.

1. Let M be a set of clauses. Then $\mathcal{C}$ is composite with respect to $M \cup N$.
2. Let $\mathcal{C}' \in N$ be composite with respect to N. Then $\mathcal{C}$ is composite with respect to $N' = N \backslash \{\mathcal{C}'\}$.

Proof:

1. Obvious from Definition 2.36.
2. $\mathcal{C}$ is composite with respect to N. The clause ordering $>_C$ is well-founded and total on ground clause. Hence, for each ground instance of $\mathcal{C}$, there exists a $>_C>_C$-minimal set P of ground instances of clauses from N that implies $\mathcal{C}$. This set does not contain a composite ground instance (otherwise we could replace it by smaller ground instances, which contradicts the minimality assumption). As $\mathcal{C}'$ is composite in N, all of its ground instances are composite as well. Ergo P does not contain a ground instance of $\mathcal{C}'$ and $\mathcal{C}$ is composite with respect to $N\backslash\mathcal{C}' = N'$ (See [BG94] for an alternative but basically equivalent proof).

■

Compositeness gives us a criterion to eliminate certain clauses. We will now extend this concept to inferences.

Definition 2.37 (Composite inference)
A generating ground inference is called composite with respect to a set N of clause if

1. one of its premises is composite with respect to N or
2. the conclusion is implied by instances of clauses from N that are smaller than the maximal premise of the inference (keep in mind that $>_C$ is total on ground clauses) or
3. it is a superposition inference into a selected positive literal.

A general inference is composite if all its ground instances are. ◀

As with compositeness for clauses, compositeness of inferences is stable against addition of clauses and deletion of composite clauses:

Theorem 2.6 (Stability of compositeness II)
Let N be a set of clauses.

1. Let M be a set of clauses. An inference that is composite with respect to N is also composite with respect to $M \cup N$.
2. Let $\mathcal{C}' \in N$ be composite with respect to $N' = N\backslash\{\mathcal{C}'\}$. An inference Π that is composite with respect to N is composite with respect to N'.

Proof:

1. Obvious from definitions 2.36 and 2.37.
2. If the inference is composite with respect to N, one of the three conditions in Definition 2.37 holds. We proceed by corresponding case analysis:

Case 1: Let $\mathcal{C}$ be the clause in the precondition that is composite with respect to N. By Theorem 2.5, it also is composite with respect to N'. Hence, Π is composite with respect to N'.

Case 2: As in the proof to Theorem 2.5, the ground instances of $\mathcal{C}'$ used in the proof of compositeness are implied by non-composite clauses in N.

Case 3: Obvious.

■

The notion of composite inferences now allows us to define sets of clauses that are closed under a certain set of inferences:

Definition 2.38 (Saturated clause sets)

- A set of clauses N is called *saturated* with respect to **SP**, if $\mathbf{SP}(N) \subseteq N$.
- A set of clauses N is called *saturated up to compositeness* with respect to **SP**, if all generating inferences with rules from **SP** are composite with respect to N.

◀

For saturated clause sets, satisfiability is decidable.

Theorem 2.7 (Satisfiability of saturated clause sets)
Let N be a set of clauses that is saturated (up to compositeness). N is unsatisfiable if and only if it contains the empty clause.

Proof: A clause set N that is saturated up to compositeness and does not contain the empty clause defines an equality interpretation that is a model of N. For details see [BG94] and consider that all generating inferences in $\mathcal{E}$ are also inferences in **SP**. For fully saturated clause sets, consider that a clause set that is saturated is also saturated up to compositeness.

■

Of course, formulae occurring in theorem proving rarely start as saturated clause sets. A theorem proving derivation tries to generate a saturated system. If certain criteria are satisfied, it can be guaranteed that such a derivation generates a saturated system at least in the limit.

Definition 2.39 (Persistent clauses, Fair SP derivation)
Let $N_0 \vdash_{\mathbf{SP}} N_1 \vdash_{\mathbf{SP}} N_2 \ldots$ be a **SP**-derivation.

- The set of *persistent clauses* or the *limit* of the derivation is defined as $N_\infty = \cup_{j\in\mathbf{N}}\cap_{i\geq j} N_i$, i.e. as the set of all clauses that are added to the set at some time, but never removed.
- The **SP**-derivation is called *fair*, if every generating inference from N_∞ is composite with respect to $\cup_{j\in\mathbf{N}}$.

◀

Theorem 2.8 (Completeness of N_∞)
The limit of a fair **SP** derivation, N_∞, is saturated up to compositeness, and all clauses in $\cup_{i\in\mathbf{N}}\backslash N_\infty$ are composite with respect to N_∞.

Proof: [BG94] gives a proof for the inference system $\mathcal{E}$ introduced in this paper. As all non-composite generating **SP**-inferences are $\mathcal{E}$-inferences, this proof carries over without modification. However, the proof requires that all contracting inferences can be modeled as *simplification* steps, i.e. by (optionally) adding some clauses implied by the precondition, followed by deletion of composite clauses. [BG94] shows this property for cases equivalent to the rules (CS), (TD), (DD), (DR) and (RP) in **SP**. It remains to be shown for (RN)[4], (ES), and (SR). In all cases we will show that the (instantiated) clauses from the conclusion are smaller than and imply the deleted clauses from the precondition.

Now consider the relevant inference rules from Definition 2.35:

(RN): The rewritten clause, $u[p \leftarrow \sigma(t)] \not\simeq v \vee R$, is obviously smaller than $u \not\simeq v \vee R$ (as $\sigma(s) > \sigma(t)$). Moreover, $u[p \leftarrow \sigma(t)] \not\simeq v \vee R$ and $s \simeq t$ imply the original clause. It is therefore sufficient to show that $\sigma(s \simeq t)$ is smaller than $u \not\simeq v \vee R$ (which implies that it is smaller for all ground instances of the affected clauses).

If $p \neq \lambda$, $\sigma(s)$ is a true subterm of u. Because $>$ is a simplification ordering, this implies that $\sigma(s) < u$. By transitivity of $<$, $\sigma(t) < u$ holds as well. Hence $\sigma(s \simeq t) <_L u \not\simeq v$.

If $p = \lambda$, $u \equiv \sigma(s)$. Therefore $\{\{\sigma(s)\}\{\sigma(t)\}\} << \{\{u[p \leftarrow \sigma(s)], v\}\}$ and ergo $\sigma(s \simeq t) <_L u[p \leftarrow \sigma(s)] \not\simeq v$.

In both cases, $u \not\simeq v \vee R >_C \sigma(s \simeq t)$.

(ES): For the case $p = \lambda$, (ES) is a special case of (CS) covered in [BG94]. For $p \neq \lambda$, $u[p \leftarrow \sigma(s)] >_{TT} \sigma(s)$ and $u[p \leftarrow \sigma(t)] >_{TT} \sigma(t)$. Again, as $>$ is a simplification ordering, $>_{TT} \subseteq >$ and hence $u[p \leftarrow \sigma(s)] \simeq v[p \leftarrow \sigma(t)] >_L \sigma(s \simeq t)$, which implies $u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R >_C \sigma(s, t \simeq)$

(SR): The case $p \neq \lambda$ is strictly analogous to the previous case. So let us assume $p = \lambda$. Then $u[p \leftarrow \sigma(s)] \not\simeq v[p \leftarrow \sigma(t)] \equiv \sigma(s \not\simeq t)$. But $\sigma(s \not\simeq t >_L \sigma(s \simeq t)$ (as $\{\{\sigma(s), \sigma(t)\}\} >> \{\{\sigma(s)\}, \{\sigma(t)\}\}$) and therefore $u[p \leftarrow \sigma(s)] \not\simeq v[p \leftarrow \sigma(t)] \vee R >_C \sigma(s \simeq t)$.

[4] [BG94] discusses rewriting of arbitrary literals, but does not allow rewriting at the top position at all.

■

Given this theorem, we can now establish the refutational completeness of **SP**, i.e. we can guarantee that a unsatisfiable formula can be proven to be unsatisfiable after at most a finite number of inferences.

Theorem 2.9 (Refutational completeness of SP)
Let $N_0 \vdash N_1 \vdash \ldots$ be a fair **SP**-derivation. The formula N_0 is unsatisfiable exactly if there exists a a i with $\square \in N_i$.

Proof: If $\square \in N_i$ for some i, N_i is unsatisfiable. As **SP** is correct, $\square$ is implied by N_0 as well. Hence, N_0 is unsatisfiable. On the other hand, if there exists no i with $\square \in N_i$, then N_∞ does not contain the empty clause either. By Theorem 2.8, it is saturated up to compositeness, and hence by Theorem 2.7 satisfiable.

■

2.8 Summary

In this chapter we have introduced the foundations for the rest of the thesis. We have established our notation for standard mathematical and theorem proving concepts used throughout the thesis and described the basic concepts of equational deduction. Finally, we introduced the superposition calculus **SP** as an extension to previously described superposition calculi. This calculus adds more flexible criteria for literal selection and allows stronger contractions as previous ones. We have established its correctness and refutational completeness. **SP** will be used as the base for the proof procedure described in Chapter 4 and realized in our implemented theorem prover E.

Chapter 3

Learning Search Control Knowledge

In this chapter we will discuss the learning cycle of a theorem prover. Many traditional machine learning algorithms have a well-defined input, use a simple learning algorithm that generates a knowledge representation, and have an application phase in which this knowledge is used. Abstraction and generalization are encapsulated into the learning algorithm. Such algorithms are applied in domains where situations can be directly mapped onto the input and the output is meaningful directly within the application domain.

For theorem provers and similar inference systems that try to learn from their own experiences, this simple model of learning and application phases is insuficient. Knowledge acquisition and application occur in different phases, and abstraction and generalization can occur in most of the individual phases.

We will discuss the process of learning and using learned knowledge using a 3-phase model.

- Experience generation and analysis
- Knowledge selection and preparation
- Knowledge application

The following sections will discuss these steps, citing existing approaches where relevant. For a more detailed overview of the literature see [DFGS99].

3.1 Experience Generation and Analysis

The first phase for each learning proof system is the generation of experiences for the system to learn from.

In theory, this experience can come from outside the proof system, either from another proof system or from a human user. However, for high-performance theorem provers both of these options are not practical. Due to the wide variety of existing theorem provers, the different calculi used and the lack of a common language to exchange search experiences, it

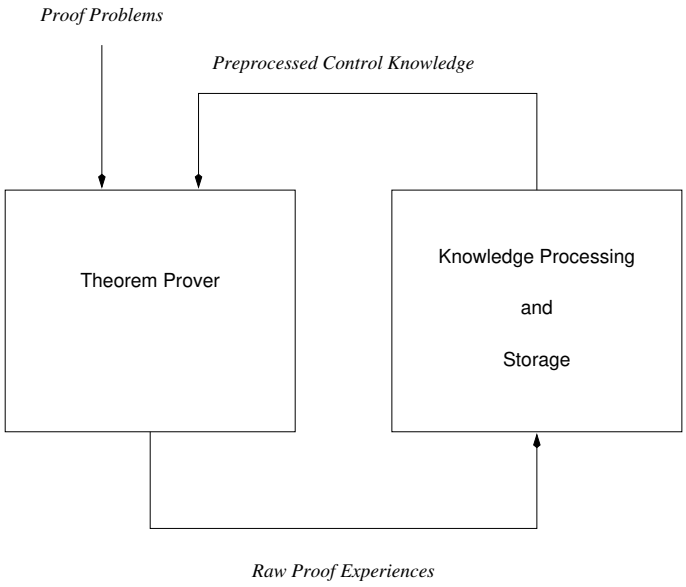


Figure 3.1: The learning cycle for theorem provers

is very hard to translate experiences from one theorem prover to another[1]. Human beings, on the other hand, are not well equipped to deal with the calculi used by theorem provers, and are quickly overwhelmed by the sheer amount of facts handled by a fully automatic proof system. Therefore, learning theorem provers are mostly limited to learn from their own experiences, i.e. the proof system itself acts as an experience generator. In this case, we have the situation of a *learning cycle* as shown in Figure 3.1.

If we consider the possible proof experiences, we can distinguish between two types of information. First, we can treat the inference machine as a black box and only consider the input, success and resource usage as relevant experiences. In this case, the proof experiences contain only *meta-knowledge* about the proof process. The second case is that we analyze the individual inferences performed by the theorem prover to arrive at its output. We call this kind of information *proof search intrinsic*.

Both kinds of information can be used for learning, and both approaches have advantages and disadvantages. Meta-knowledge is typically easy to obtain from any existing theorem prover. As the system is considered as a black box, no modifications of the inference engine are necessary. Moreover, meta-knowledge typically is very compact. Specifications for clausal proof problems are usually small, and parameter sets, result status and resource usage are even more compact. As a result, storage, analysis and processing of this knowledge is relatively easy, and often can even be performed manually or semi-automatically. The major disadvantage of these kind of experiences is the extremely strong abstraction

[1]This may, to some degree, change in the future, as there are approaches to describe proofs generated by different provers and even different calculi in a common format, as e.g. the *block calculus* [DW94] used in ILF.

resulting from the black box approach. As only outside behaviour is observed, we can only expect to learn knowledge about the relationship between proof problems and good parameters for the theorem prover. We cannot expect to learn totally new proof strategies.

Meta-knowledge approaches are widely used for the automatic configuration of theorem provers. Often, the proof problem is reduced to a set of numeric features, and feature based distance measures are used to implement a case-based reasoning approach for the selection of a strategy or set of strategies for the theorem prover. See Section 5.1 for a short survey of this technique. Alternatively, similarity measures based on the structure of the axiom set are used in a similar way (see e.g. [DFF97] or [HJL99]). In most cases, these techniques involve manual analysis of the results, however, there also are some successful attempts to automate this process ([Fuc97a],[Wol98b, Wol99b]).

Proof-intrinsic knowledge, on the other hand, is much harder to obtain. A potential proof experience in this case is a sequence of inference steps, describing the complete proof derivation. To obtain this sequence, most existing theorem provers have to be modified significantly. Moreover, the amount of data we have to handle in this case is enormous. A typical proof derivation for a non-trivial problem contains between several thousand and several million inferences. However, the potential rewards also increase proportionally. Since we can analyze the proof process at an inference level, we can hope to find completely new proof strategies that can help the prover to solve problems that none of its existing search strategies can successfully tackle.

In the case of proof-intrinsic knowledge, we typically have to reduce the amount of data to a manageable amount. This is achieved by *proof search analysis*. Such an analysis tries to reduce the total number of inferences (or the search decisions represented by these inferences) down to a subset of particularly significant events. The degree of abstraction at this stage varies widely:

- Of course the most general representation is to perform no abstraction at all. However, we do not consider this approach to be practicable, and we know of no recent attempts to use it.

- The next degree of abstraction tries to reduce the amount of data by concentrating on inferences that are part of the proof or close to the proof. We describe the necessary techniques for saturating theorem provers in some detail in Section 5.3. Most theorem provers that try to learn *heuristic evaluation functions* use these or similar, more ad-hoc, techniques to generate training examples. SETHEO/NN ([Gol99a, Gol99b]) uses tableaux representing proofs, and tableaux derivable from those in a few inferences, as examples. Some of the learning approaches integrated into DISCOUNT ([Fuc96, Fuc97b, FF98] and [DS98, Sch98] use processed facts (rules or equations) to represent search decisions.

- Even more abstraction occurs if one considers only inferences or facts actually contributing to the proof. This is used in analogy-based approaches like *flexible reenactment* [Fuc96, Fuc97b] and the algorithms based on derivational analogy that have been applied in inductive theorem proving (see e.g. [MW96, MW97b]). We have used

this approach for learning evaluation functions as well, see [DS96a, DS98, SB99], although we annotate the selected facts with information about their role in the global proof process.

- As a borderline case to the meta-knowledge approach, we can reduce the proof experience even further and just keep the subset of (potentially instantiated) original axioms necessary to find the proof. This approach is taken by learning approaches inspired by *explanation-based generalization* as e.g. described in [KW94], where this kind of information is collected in a generalized *proof catch*.

In addition to the type of proof experiences, we have to discuss the *selection* of proof experiences from the usually infinite space of all proof derivations. Predicate calculus can be used to encode nearly arbitrary problems, and it is possible to construct formulae that force very untypical proof derivations. A very natural way to restrict the set of training examples therefore is to demand that the problem specification is taken from some domain of interest to humans.

Similarly, there are typically a lot more unsuccessful than successful search derivations for each given resource limit. Therefore it is reasonable to primarily use positive examples, i.e. representations of successful proof searches.

Finally, it is possible to restrict the search experiences by resource limit, i.e. to only use search derivations that can be derived within certain resource bounds.

3.2 Knowledge Selection and Preparation

The second phase of learning involves the selection of suitable knowledge from the set of all experiences and the preparation of this knowledge in a way that aids the application. At the core of this phase we typically find one or more traditional machine learning algorithms. We give a more detailed discussions of these algorithms in Section 6.1 for term-based algorithms and at the end of Section 5.1 for feature-based approaches.

Selection and preparation can in principle happen in arbitrary order. A theorem prover can store pre-processed knowledge and select an appropriate part of this knowledge as it becomes necessary, or it can select raw proof experiences and prepare them on demand.

Provers that store pre-processed knowledge typically use relatively slow learning algorithms. Examples are different types of neural networks ([SE90, Gol94],[Gol99a, Gol99b]) or genetic algorithms [Fuc95a]. All known existing implementations leave the selection of suitable knowledge to the user. The prover is optimized for use in a single domain, and no automatic mechanism for switching to different classes of problems is provided.

Systems with fast learning algorithms typically select knowledge based on the problem at hand. They usually perform some analysis of the new problem specification and then collect information from experiences with *similar* problems, i.e. they use *case-based learning* (see e.g. [Kol92]). Similarity is either based on numerical representations of the proof problem (see Section 5.3) or directly uses techniques to compare the structure of the problem formula. In this case, either various versions of *matching* between parts of the new formula

and existing proof experiences are used to determine similarity (see e.g. [DS96a, DS98] and the approaches described for meta-knowledge in the previous section), or numerical similarity measures are defined directly on the formulae [SB99].

Fast learning algorithms used in theorem proving often are very simple. Even plain memorization of important facts can improve the performance of theorem provers, because generalization occurs in the analysis and in the application phases. In addition to the memorization of facts (used e.g. in flexible reenactment [Fuc96, Fuc97b]) we have memorization of generalized *patterns* [DS96a, DS98] with evaluations, and recursive *term evaluation trees* [DS96a, DS98]. As an extension of term evaluation trees, we develop general *term space maps* in this thesis. Some instances of these term space maps have already been described in [SB99, Sch98].

A special case is that knowledge selection, knowledge preparation and knowledge application are combined into a single step. This happens in the case of proof reuse, where the selection involves (first- or second order) matching of the axioms, but where a successful match immediately yields a complete proof for a (sub-) problem.

3.3 Knowledge Application

The final phase is the application of learned and prepared knowledge. For the special case discussed above, this is trivial and typically only requires the substitution of the goal with new instantiated subgoals. For the case of meta-knowledge, this is achieved by simply starting the prover with appropriate parameter settings.

The most complex case is that the search decisions are dynamically influenced by the learned knowledge. We have two sub-cases to consider: Analogical replay and heuristic evaluation (with some approaches in between these two extremes).

In the case of analogical replay, the prover uses a source proof to guide the search. It tries to match the current proof situation onto a situation in the source proof search and selects the inference that led to a success. If no matching situation is found, the prover needs to *patch* the proof search in some way, typically by performing a (blind or heuristic) search.

In the case of heuristic evaluation, a learned evaluation function evaluates alternatives in the proof search. The prover then performs the inference with the best evaluation. Typically, learned knowledge is combined with a standard heuristic to allow for graceful degradation in the case that the learned heuristic does not cover parts of the search space.

An in-between case is *flexible reenactment* (see above), which is inspired by derivational analogy [Car86, CV88]. Instead of mapping situations in the target proof search onto the source proof search, a global evaluation function is used. Search decisions are represented as clauses to be processed, and clauses useful in the source problem are preferred in the target proof search. *Patching* is provided by a conventional backup strategy and improved by having clauses not recognized inherit part of the good evaluation of their ancestors.

3.4 Our Approach

As we wrote in Chapter 1, our aim is to use learning techniques both to adapt the theorem prover to a given domain and to improve its overall performance. To achieve this aim, we develop a learning theorem prover that offers solutions for all of the above phases. Our central learning algorithms use proof-intrinsic knowledge, meta-data is used for the selection of suitable proof experiences.

Our approach is a continuation of the earlier work done by Fuchs [Fuc95a, Fuc96, Fuc97b] and ourselves [Sch95, DS96a, DS98] for the unit-equational case. We represent important search decisions as individual clauses and try to learn good heuristic evaluation functions from example clauses taken from successful proof searches.

In the next chapter, we develop an efficient algorithm for superposition-based theorem proving. We identify the different choice points and discuss the resulting search problem.

Our approach to the analysis phase is described in Chapter 5. We describe a proof problem using numerical features (for experience selection) and sets of annotated *clause patterns* from clauses close to the actual proof to represent search decisions.

These annotated clause patterns are used as the input for a class of new learning algorithms described in Chapter 6. *Term space mapping* works by partitioning the set of all terms into a finite number of subsets (either once or repeatedly) and by associating an evaluation with each class. Both this partitioning and the evaluations are based on the distribution of terms and evaluations in a training set. The *term space maps* constructed by these algorithms define a heuristic evaluation function that is used to guide the proof search.

Chapter 7 describes the overall system and the interaction of the different components.

Chapter 4

Search Control in Superposition-Based Theorem Proving

The unsatisfiability of a clausal formula is, in general, undecidable. All algorithms for theorem proving therefore have to *search* for a proof in a usually infinite space. In this chapter we will analyze the theorem proving process of a superposition-based theorem prover from this point of view.

We first introduce an abstract model for discrete search problems of the kind occurring in automated theorem proving. We show different ways to map standard theorem proving algorithms onto this model and discuss which search decisions have to be made before and during the proof search for a superposition-based theorem prover. We describe the *given-clause* algorithm and develop a variant of it for the superposition calculus. With this algorithm we eliminate certain choice points and explain the rationale underlying these decisions. We also develop a sufficient condition for the completeness of theorem proving derivations generated by this algorithm.

Then we discuss the remaining choice points, their influence in the proof process, and how they are typically controlled in a conventional saturating theorem prover. We identify the selection of the next clause to process as a critical choice point and as the most suitable choice point to be controlled by search control knowledge gained from the analysis of other proof searches. We conclude this chapter with a survey of conventional evaluation heuristics for the control of this choice point.

4.1 The Search Problem

A general *search problem* is described by a set of *search states* and a *transition relation* that describes which search states can be reached from any given other state:

Definition 4.1 (Search problem)
Let M be a set (of *search states*), let E be a binary *transition relation* on M and let

$w : E \to \mathbf{R}$ be a weight function. Then the weighted directed graph $T = (M, E, w)$ is called a *search space*. Let $s \in M$ be a *start state* and let $G \subseteq M$ be a set of *goal states*.

- $P = (T, s, G)$ is called a *(discrete) search problem*.
- The set of *search paths* for P is $\mathcal{S}(P) = \{p \in \mathcal{P}_T | p = s.p', p' \in \mathcal{P}_T\}$.
- A search path $s, s_1, \ldots, s_n$ is called *successful* or *a solution for the search problem* if $s_n \in G$.
- The *cost* of a search path is $cost(s_0, \ldots, s_n) = \sum_{i=0}^{n-1} w((s_i, s_{i+1}))$.
- If $s \in M$ is a search state, then $|succ(s)|$ is called the *branching factor* of the search space at this state.

◀

A search path describes a single set of choices starting at an initial state and hopefully reaching a goal state. However, during the search, we may need to deal with a multitude of such paths.

Definition 4.2 (Search derivation)
Let $p = s_0, \ldots, s_n$ be a search path. We define three operations (with associated cost) on p:

Extension: $s_0, \ldots, s_n \Vdash_E s_0, \ldots, s_n, s$ for $s \in succ(s_n)$. The cost of an extension step is $cost(s_0, \ldots, s_n \Vdash_E s_0, \ldots, s_n, s) = w((s_n, s))$.

Backtrack: $s_0, \ldots, s_n \Vdash_B s_0, \ldots, s_{n-1}$ if $n > 0$. The cost of a backtrack step is 0[1].

Startover: $s_0, \ldots, s_n \Vdash_S s_0$. The cost of a startover step is 0.

The *search derivation relation* $\Vdash$ is $\Vdash_E \cup \Vdash_B \cup \Vdash_E$.

- A *search derivation* D is a sequence of search paths $p_0 \Vdash \ldots \Vdash p_n$. The set of all search derivations for a search problem P is D_P.
- The cost of a search derivation is the cost of the extension steps performed during the derivation:

$$cost(p_0 \Vdash \ldots \Vdash p_n) = \sum_{i=0}^{n-1} cost(p_i \Vdash p_{i+1})$$

[1]In practice, cost for backtracking (not to be confused with the total cost spent in backtracked search alternatives) is often negligible. Moreover, cost for backtracking can be easily included in the cost for the corresponding extension step.

- A search derivation $p_0 \Vdash \ldots \Vdash p_n$ for a search problem P is *successful within a given cost bound* b, if there exists an $i \in \{0, \ldots, n\}$ so that p_i is a solution to P and $cost(p_0 \Vdash \ldots \Vdash p_i) \leq b$.

◀

A search strategy is a function that generates a search derivation. It defines a successor state for each finite search path and thus inductively a complete search derivation.

Definition 4.3 (Search strategy)
Let P be a search problem.

- A function $S : D_P \rightarrow M \cup \{Backtrack, Startover\}$ with the property that for all derivations $D = p_0 \Vdash \ldots \Vdash p_n$ where $p_n \equiv s_0, \ldots, s_n$, $S(D) \in succ(s_n) \cup \{Backtrack, Startover\}$ is called *a search strategy* for P.
- It defines a search derivation $D(S)$ in the obvious way.
- A search strategy is *successful within a given cost bound* if the corresponding search derivation is.
- A search strategy is *complete* if it eventually finds a solution whenever a solution exists.

◀

This model of search processes is general enough to describe the search problem for most theorem proving procedures.

Example: Consider the case of a proof procedure for a model-elimination prover [Lov68], as e.g. SETHEO [LSBB92, MIL+97]. The proof problem for a set $\mathcal{F}$ of clauses is given by $P = ((M, E, w), s, G)$ as follows:

- M is the set of all connection tableaux for F
- $E = E_S \cup E_R \cup E_E$ with
  - $E_S = \{(\nabla, \mathcal{C}) | \nabla$ is the empty tableau, $\mathcal{C} \in \mathcal{F}\}$
  - $E_R = \{T, \sigma(T)) | \sigma(T)$ is the result of a tableaux reduction step $\}$
  - $E_E = \{(T, T') | T'$ is the result of a tableaux extension step with T and a clause from F $\}$
- There are of course many possible cost functions w. A possible example measures the number of unifications:

$$w((T, T')) = \begin{cases} 0 & \text{if } (T, T') \in E_S \\ 1 & \text{otherwise} \end{cases}$$

For the usually more interesting case of execution time we need to know the details of a particular implementation down to the hardware. The cost of a particular inference step will in this case include the cost for unification, changes in the tableaux, and the local search for the next possible inference, and can usually be approximated as a function of size and depth of the resulting tableaux and the number of literals in clauses from $\mathcal{F}$.

- $G = \{T|\text{T is a closed tableaux}\}$
- $s = \nabla$

The typical iterative deepening search procedure enumerates all search paths (sequences of possible tableaux) up to a certain length, using *extension* and *backtracking*, and would then use a *startover* step and repeat the procedure with a larger length limit.

If we want to map the search in superposition-based theorem proving to this framework, we have the choice between various mappings. However, if we assume a given term ordering, we get a very natural mapping for the most general case:

- A single search state corresponds to a a set of clauses. In other words, $M = 2^{Clauses(F,P,V)}$.
- The transition relation is described by the inference system $\mathcal{SP}$ introduced in section 2.7. More exactly, $(s, s') \in E$ iff $s \vdash_{\mathbf{SP}} s'$.
- The most interesting cost measure is again the time a certain inference takes in a given implementation. However, this depends on details that, due to the large complexity, are not generally accessible to a theoretical analysis[2]. A very simple approximation ignores the search for possible inferences and assigns a fixed weight to each transition, i.e. $w(s, s') = 1$ for all w. If we incorporate the cost for evaluating *all possible inferences*, a lower bound for the cost is certainly given by $n \times n$, as we need to consider each pair of clauses (and in fact, need to consider much more than one potential inference position within each clause). While indexing techniques (as e.g. presented in [Gra95, GF98]) can reduce the number of candidates for an inference, it can never reduce the number of possible inferences. In practice, this number even seems to grow exponentially with the size of the clause set (compare the discussion and experimental results is section 4.3).
- The set of goal states is $G = \{s \in 2^{Clauses(F,P,V)} | \square \in s\}$, i.e. the set of clause sets containing the empty clause.

[2]As an example, the widespread use of term indexing techniques has reduced the cost for finding a rewrite rule applicable at a given term position in a way that this cost is usually irrelevant compared to other operations. For equational provers using linear search this operation is a major cost factor.

- Finally, the start state is the initial set of clauses from the problem specification.

A sufficient condition for the completeness of a search strategy for the superposition calculus is the fairness of the corresponding **SP**-derivation. However, as fairness only requires that all generating inferences are *composite* in the limit, any finite **SP**-derivation can be continued to a fair one. This leads to the following corollary:

Corollary: Backtracking steps and startover steps are not necessary for a complete search strategy in the superposition calculus **SP**.

In practice, backtracking steps are extremely rare in saturating theorem provers, although they can be useful if some analytical features are integrated into the prover. One example is SPASS [WGR96, WAB+99], which extends the superposition calculus with a *splitting*-rule for clauses. Clauses generated during a split possibly need to be retracted later on.

Startover steps, on the other hand, have recently become quite popular for fully automatic, self-configuring systems. The best example for a purely saturating theorem prover employing startover is Gandalf [Tam97, Tam98]. Gandalf sequentially tries a number of different search strategies up to a certain cost limit. Similar composite strategies are used by the hybrid theorem prover p-SETHEO [Wol98b, WL99, Wol99a], which selects a given schedule incorporating many different individual theorem proving strategies.

4.2 Proof Procedure and Choice Points

The very high degree of non-determinism allowed by the constraints of the inference system and the fairness condition is very hard to manage. As an example, for the fairly conservative estimate of 100,000 clauses with two maximal terms in eligible literals and an average of 10 term position in each term, we have to consider approximately 400,000,000,000 potential paramodulations and about 5,000,000,000 potential subsumption steps even if using some simple pruning techniques. Therefore, nearly all successful saturating theorem provers restrict most of the choice points by using the *given-clause* algorithm first popularized by Otter and used (with slight variation) in most current saturating theorem provers, including e.g. DISCOUNT [DKS97], Gandalf [Tam97, Tam98] SPASS [WGR96, WAB+99], Vampire [RV99] and Waldmeister [HBF96, HJL99]. We also use a variant of this algorithm in our own theorem prover, E.

In the given-clause algorithm, the control over generating inferences is simplified by splitting the set of all clauses into a subset of processed clauses and a subset of unprocessed clauses. At each state on the proof process, all generating inferences between clauses in the processed subset have already been performed. A *given clause* is moved from the set of unprocessed clauses to the set of processed clauses by computing all generating inferences which involve the given clause and clauses from the processed set. The second important

feature responsible for the success of this algorithm is the preference of contracting inferences over generating ones. Clauses are only used for generating inferences if they are not redundant with respect to the processed clause set (or the newly selected clause), and if they cannot be simplified further. Figure 4.1 shows a sketch of the algorithm.

Variables:

`C`	Set of *unprocessed* clauses, contains initial clauses at start of algorithm
`S`	Set of selected and *processed clauses*, initially empty
`c`	The *given clause*, focus of all inferences for a given execution of the main loop

```
while C is not empty
{
   pick c from C (in a fair manner);
   perform all contracting inferences
   with clauses from S on c;
   if c is the empty clause then success, proof found;
   if c is not trivial or subsumed
   {
      perform all contracting inferences with c
      on clauses from S, moving affected clauses into C;
      perform all generating inferences between c
      and clauses from S, putting new clauses into C;
   }
}
failure, no proof found;
```

Figure 4.1: The *given-clause* algorithm

The given-clause algorithm simplifies the control problem and reduces the problem of inference selection in various ways:

- The invariant (all generating inferences between processed clauses have been performed, the processed clause set is in a normal form with respect to the contracting inferences) makes a more detailed administration of possible inferences unnecessary.
- Contracting inferences are primarily performed using a small subset of clauses. This subset changes only relatively seldom and usually in a small way. Therefore it is possible to compile this set for more efficient operations. Typically, some kind of indexing is used to find potential inference partners more efficiently.

- As all generating inferences with the given clause and the processed clause set are performed at once, the time for searching potential inference position is shared for all these inferences. It is not necessary to reconsider all potential inferences at each stage in the search process.

- The branching factor at each stage of the search drops significantly, making the both the decisions during the search and their implementation much easier. Instead of selecting one of all possible inferences, the most difficult choice point now is the selection of the *given clause*. For typical cases, this reduces the number of possible decisions by multiple orders of magnitude[3].

The general algorithm in Figure 4.1 can be refined for the superposition calculus **SP**, fixing choices for further choice points in the algorithm. In particular, while the order in which generating inferences are performed for each given clause is not important, the order of contracting inferences can seriously influence the performance of the prover. Moreover, for this more specific case, additional optimizations are possible. Figure 4.2 shows a refined procedure for the superposition calculus, as implemented by E. The subroutines are explained in Figure 4.3.

Again, we have removed or simplified a number of choice points using pragmatic reasoning. Current theorem proving technology makes rewriting (and recognizing rewritable terms) relatively cheap, while testing for subsumption remains expensive for non-unit clauses. Moreover, the chance for a successful subsumption is increased if all affected clauses are in normal form with respect to the same system of unit-clauses. Therefore rewriting and clause normalization, for which similar arguments hold, are performed before subsumption.

The remaining choice points have been encapsulated in the subroutines described in Figure 4.3. We will now discuss these choice points and the strategies employed at these choice points in some detail. Note that `normalize(c,S)` and `max_rw(S,c)` do not contain critical choice points. While there remains some non-determinism about *how* they perform their respective operations, the result is fully determined by the input. Similarly, the order in which generating inferences are performed in `generate(c,S)` is not critical, although the question *which* inferences are performed is.

The remaining choices are the selection of a term ordering (which we have considered fixed up to now), the rewriting strategy, the literal selection strategy and finally the clause selection strategy that is encapsulated in the function `select_best(C)`.

As this last choice point controls the order of generating inferences, it is the only choice point that influences the theoretical completeness of the theorem proving procedure. The definition of fairness, Definition 2.39, requires that all generating inferences between

[3] An alternative view is to consider only the set of processed clauses as the proof state and to consider the set of unprocessed clauses only as a *preview* of possible (generating) inferences. In this case the branching factor is only reduced by the strict ordering of different inference types. However, the main cost of the proof search is caused by inferences involving unprocessed clauses, and hence we consider our view to be more helpful for this analysis.

Variables:

`C`	Set of *unprocessed* clauses, as in Fig 4.1
`S`	Set of *processed clauses*
`T`	Temporary store for newly generated clauses
`c`	The *given clause*
`c'`	Temporary handle for clauses

```
while C ≠ ∅
{
  c := select_best(C);
  C := C \ {c};
  c := normal_form(c,S);                                    (RN),(RP)
  c := normalize(c,S);                                   (SR),(DD),(DR)
  if c = □ then success; end
  if c is not tautological then                                  (TD)
  {
    if is not subsumed by S then                            (ES),(CS)
    {
      T := max_rw(S,c);
      S := S \ T;
      S := S \ subsumed(c, S);                              (ES),(CS)
      S := S ∪ {c};
      S := interreduce(S);                                   (RN)(RP)
      T := T ∪ generate(c,S)                             (SN)(SP)(ER)(EF)
      forall c' ∈ T
      {
        c' := normal_form(c', S);                           (RN),(RP)
        c' := normalize(c', S);                          (SR),(DD),(DR)
        if c' is not tautological then C := C ∪ c';              (TD)
      }
    }
  }
  failure; no proof found;
```

The two-letter codes to the right refer to **SP**-inference rules, compare Definition 2.35. For explanations of the subroutines see table 4.3, page 45.

Figure 4.2: A *given-clause*-derived algorithm for superposition

`select_best(C)`	Return the best (with respect to some heuristic) clause from `C`.
`normal_form(c,S)`	Compute the normal form of `c` with respect to the positive unit clauses in `S`.
`normalize(c,S)`	Remove superfluous literals from the clause `c`, first by applying *simplify-reflect* and then by removing duplicate and resolved literals.
`max_rw(S, c)`	Return the clauses from `S` in which a maximal term in a eligible literal can be rewritten with the clause `c`.
`subsumed(c, S)`	Return the clauses from `S` which are subsumed by `c`.
`interreduce(S)`	Interreduce the system of clauses `S`, i.e. reduce all clauses `c` in `S` to a normal form with respect to $\mathtt{S} \setminus \{\mathtt{c}\}$.
`generate(c,S`	Compute the set of all clauses that can be deduced with generating inferences (superposition, equality factoring and equality resolution) where `c` is at least one of the clauses in the precondition and the remaining clauses in the precondition are elements of `S`.

Figure 4.3: Subroutines for the *given-clause* procedure in Figure 4.2

persisting clauses are composite with respect to the set of all clauses occurring in the derivation. Theorem 2.6 implies that a sufficient condition is that all generating inferences are composite with respect to some intermediate clause set. We will now show a sufficient condition for the clause selection function to ensure this.

Theorem 4.1 (Fairness of given-clause proof derivations)
The proof derivation generated by a given-clause algorithm is fair, if no clause remains in the set `C` forever.

Proof: We have to show that all generating inferences between persisting clauses are composite with respect to some intermediate clause set. We will show that if $N \vdash_{\mathbf{SP}} N'$ with a generating inference, then this inference is composite with respect to N' (*). Given this result, assume that $\mathcal{C}$ and $\mathcal{C}'$ are two arbitrary persistent clauses. By assumption, both are removed from `C` at some time, and put into `S`. But the invariant of the given-clause algorithm is that all generating inferences between clauses in `S` have been performed. Hence, all generating inferences between $\mathcal{C}$ and $\mathcal{C}'$ will be performed at some time. As this holds for arbitrary clauses $\mathcal{C}$ and $\mathcal{C}'$, all generating inferences between persisting clauses will be performed and by (*), the resulting derivation is fair.

We now will show the claim (*). By definition, a generating inference is composite if all its ground instances are composite. We show that an arbitrary ground inference is composite with respect to its conclusion. To do this, we show that the conclusion of any ground inference is $>_C$-smaller than the maximal premise (compare Definition 2.37).

- Consider the case of a ground (ER) inference[4]:
  $$\frac{u \not\simeq u \vee R}{R}$$
  Clearly, $u \not\simeq u \vee R >_C R$.
- Now consider the case of a ground (SN) inference:
  $$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{u[p \leftarrow t] \not\simeq v \vee S \vee R}$$
  Since $>$, $>_L$ and $>_C$ are total on ground terms, literals and clauses, s is the strictly maximal term in the first premise. Moreover, since s is a subterm of u and $>$ is a simplification ordering, u is larger than all other terms in the first premise. Moreover, $u > v$ and $u > u[p \leftarrow t]$. Therefore the literal $u \not\simeq v$ is larger than all any literal in S and larger than $u[p \leftarrow t \not\simeq v]$. Ergo the conclusion is smaller then the second premise.
- The case of a (SP) inference is strictly analogous.
- Finally, consider a (EF) inference:
  $$\frac{s \simeq t \vee s \simeq v \vee R}{\sigma(t \not\simeq v \vee s \simeq v \vee R)}$$
  The ordering constraints imply that s is a maximal term in the clause. Thus, $s \simeq t >_L t \not\simeq v$ and hence the precondition is larger than the conclusion.

■

4.2.1 Term orderings

So far, we have assumed a fixed term ordering. However, the selection of a suitable term ordering for the proof process is an additional choice point that can be vital for the success of the search. Especially in the case of unit-equational problems this is probably one of the most important decisions.

The superposition calculus assumes a single fixed term ordering for the complete proof search. While it is possible to construct this ordering during the early stages of the proof search, most fully automatic high-performance theorem provers either rely on a user-specified term ordering or select an ordering at the very beginning.

The Waldmeister system has the most elaborate ordering selection system of all current high-performance theorem provers [HJL99]. Waldmeister selects a suitable term ordering based on an automatic detection of the domain of the proof problem, i.e. by matching the

[4]Keep in mind that in the ground case terms are unifiable exactly if they are identical, and no substitutions are applied.

axioms against an internal database of problems with associated orderings. The demonstrated superiority of Waldmeister for unit-equality problems suggests that this approach is very adequate.

Construction of such a database, either based on axiom matching or on feature-based similarity measures, is a fairly straightforward application for case-based learning techniques. We plan to incorporate such a system into future versions of our theorem prover E. However, this approach calls for a meta-information based approach to learning: Instead of analyzing individual proof searches in detail, it is necessary to analyze only the *performance* of different proof searches for the same problem or a class of problems. Therefore, we will deal with this problem in a separate work.

4.2.2 Rewriting strategy

The procedure `normal_form(c,S)` encapsulates the rewriting part of the proof procedure. A very general algorithm for this subroutine (for terms) is given in Figure 4.4 – for larger structures like literals or clauses the same algorithm is iteratively applied to all terms in the structure. There are two choices involved for each rewriting step:

- Selecting a position at which to rewrite
- Determining which of the matching unit clauses to apply

Any normal form procedure has to ensure that *all* subterms are irreducible. Experimental results, on the other hand, show that most subterms generated during theorem proving are top-irreducible. Therefore algorithms that try to select a rewrite position from the set of all possible positions are at a serious disadvantage, as the overhead of collecting all possible positions at each rewrite step is relatively high.

Thus, most existing theorem provers use a fixed term traversal strategy[5] (with backtracking). While in principle arbitrary strategies are possible, the three most easily organized ones are *innermost*, *outermost* and *breadth-first top-down*. The *innermost* rewrite strategy traverses the term in post-order, i.e. it rewrites subterms (either in left-to-right or in right-to-left order) first, then the super-term. The *outermost* strategy implements pre-order traversal, i.e. it checks the top term first and then descends to the arguments. Finally, *breadth-first top-down* rewriting visits all nodes ordered by their depth in the path (and with an arbitrary but fixed order within each level).

[BH96] contains a discussion and experimental evaluation of the three traversal strategies for the unit-equational theorem prover Waldmeister. The authors conclude that no traversal strategy is optimal in all cases, and the traversal strategy has to be tailored to the data structure of the term representation. For Waldmeister, the conclusion reached in

[5]Recent versions of the Waldmeister theorem prover (which is based on unfailing completion and hence has only a single unit-clause as the goal) do compute *all* normal forms of terms in a goal [HJL99]. This is not feasible for a more general theorem prover, and and even in the Waldmeister case the number of successors for the goal has to be artificially limited for some proof problems.

Variables:

S	Set of clauses describing the rewrite-relation
c	A single clause
t	Term to be rewritten
S'	Set of positive unit-clauses from S
p	Position in t

```
procedure normal_form(c,S)
{
  S' := {c ∈ S | c is a positive unit clause};
  while t is S'-reducible
  {
    Select p so that t|_p is S'-top-reducible;
    Select c≡l=r from S' so that σ(l)=t|_p and σ(l)>σ(r);
    Replace t|_p with σ(r);
  }
  return t;
}
```

Figure 4.4: A generic normal form algorithm

the original report was that the *outermost* strategy corresponds best to the flat term representation implemented in this system, however, later analysis uncovered some flaws in the implementation used for the evaluation of the *innermost* strategy. The preliminary revised results seem to show that the differences between innermost and outermost term traversal show only in rare examples, in which case either strategy can have advantages [Löc99].

The selection of a rewrite strategy based either on the term to be rewritten or general problem characteristics is a possible choice point where learning can be employed. However, the influence of this choice point is probably restricted to problems where rewriting plays a key role, and seems to have less overall influence than other choice points. We have therefore decided to implement a fixed standard solution. As our own prover is build on a shared-term rewrite engine, *innermost* is the most obvious choice (and the only one that can be consistently implemented for all terms), as it allows maximal benefit from shared rewriting and can be optimized using normal form dates on terms. Appendix A.1.1 discusses the advantages of this choice.

The second choice is *which* rules or equations to try at each term position. There is no detailed evaluation of this choice point which we are aware of. Typically, the search for a reducible position and the search for an applicable unit clause are combined, i.e. all rules or equations are tried at each term position. In this case, an obvious point (again supported by experiments in [BH96]) is that rules (i.e. unit clauses in which one term is larger than the other in the term ordering) should be tried first. Equations can only be used for rewriting if the instantiated equation is orientable (in the desired direction). Therefore, a relatively expensive ordering test has to be performed for each instantiation of the clauses (which quite often fails). For rules, on the other hand, only a single test is necessary when the rule is created. Apart from this optimization, most current provers leave the choice of the rule up to the convenience of the implementation, i.e. they traverse a linear list or a indexing tree structure and use the first applicable rule.

As the number of applicable rules at each term node typically is small (remember that the set of rules and equations is interreduced), the impact of a particular solution is, in most cases, quite minimal. In E, we have implemented both linear lists and perfect discrimination trees ([Gra95, GF98], also compare Appendix A.1.2) with two different traversal strategies (more general rules first or more special rules first), and have found only slight differences between both tree-based versions. The version based on linear traversal of the clause list is, of course, a lot slower, but otherwise behaves fairly similar as well. Table 4.1 shows the relevant data for three typical unit problems we use for illustration (see Appendix B).

A final choice point related to rewriting is the order in which positive unit-clauses are selected for being rewritten during interreduction. Again, successful rewriting during interreduction is a fairly rare operation. Moreover, it will not influence maximal terms. We take this as an indication that this choice-point is of little practical relevance and thus have opted for the most convenient solution. That means that clauses are considered in the natural order, i.e. the oldest clause in the set of processed positive unit clauses is considered first.

As the total influence of the rewriting strategy seems to be limited, the application of learning techniques to control is unlikely to result in drastic improvements. Moreover, as for the case of term orderings discussed above, any single proof search generated by a standard theorem prover only contains information about a single strategy. Learning, thus, would again require the analysis of multiple proof attempts or significant changes to the inference engine to enable the prover to explore different possibilities in parallel.

4.2.3 Clause Selection

The selection of the next clause to process (i.e. the selection of the next *given clause*) is the most important choice point for *given-clause* based theorem provers. Even for hard proofs, only a relatively small and easily manageable number of clauses actually participates in the proof. Table 4.2 shows the numbers for our standard set of examples and a fixed strategy.

As the table shows, while the number of generated clauses range over more than three orders of magnitude, and the number of processed clauses range over nearly three orders of magnitude, the number of clauses in the proof actually range over only about one order

Problem/Strategy	Processed clauses	Non-trivial	Time
INVCOM			
General fist (PDT)	15	14	0.05 s
Specific first (PDT)	15	14	0.04 s
Oldest first (linear list)	15	14	0.03 s
BOO007-2			
General fist (PDT)	4454	3185	23.34 s
Specific first (PDT)	4489	3220	22.38 s
Oldest first (linear list)	4829	3560	103.63 s
LUSK6			
General fist (PDT)	3673	3103	20.31 s
Specific first (PDT)	3672	3103	19.92 s
Oldest first (linear list)	4894	4270	76.85 s

Remarks: Shown are the number of clauses processed (i.e. selected as *given clause)*, the number of these clauses that were non-trivial after rewriting, and the time for the search until a proof has been found. Times are measured on a SUN Ultra 10/300. Times for INVCOM are of the same order of magnitude as the resolution of the timing command, and differences there are not significant. Strategies marked with (PDT) use a perfect discrimination trees. Term ordering was the standard KBO (see A.1.3), clause selection was according to the `Weight` heuristic described in A.2.1.

Table 4.1: Selection of rewriting clause

Problem	Generated clauses	Processed clauses	Proof clauses
INVCOM	129	21	11
BOO007-2	198372	10124	52
LUSK6	55196	3672	108
HEN011-3	341044	4813	130
PUZ031-1	120	107	48
SET103-6	91887	4544	15

Remarks: Results are given for the `StandardWeight` clause selection heuristic and the default term ordering. In all clauses with at least one negative literal, the largest negative literal was selected.

Table 4.2: Generated, selected and useful clauses

of magnitude. In fact, even if we check a much larger set of examples, there is rarely a proof that needs more than 200 clauses. If we compare this to the total number of clauses processed in the above table, it is obvious that the amount of work associated with this number of clauses typically is very small by todays standards. If a prover picks the right clauses, all proofs we have encountered so far can be reproduced in less than 10 seconds

even for large proofs, and more typically in less than 3 seconds on standard hardware.

The selection of the next clause to process has a decisive influence for nearly all classes of problems encountered. The proof search for unit problems, Horn problems, and general problems depends critically on good clause selection. The choice point also is of equal importance for problems with and without equality. For these reasons, we consider this choice point to be most suitable for control through our our learning approach. As an added benefit, this choice point is the only choice point where examples of good and bad search decisions can be learned from a single proof search, as clauses can be clearly separated into useful and superfluous clauses.

As this choice point is important for nearly all saturating theorem provers regardless of the implemented calculus, it is also the choice point that most work has been done on. We discuss the existing techniques in more detail in section 4.3, where we also include some data on the performance of simple clause selection schemes. [DF98] contains a detailed discussion and experimental comparison of this choice point for the theorem prover DISCOUNT.

4.2.4 Literal selection

The procedure `generate(c,S)` encapsulates all applications of the generating inference steps (ER), (SN), (SP) and (EF). The only relevant choice point is the selection of a literal selection function for the new clause `c`. Literal selection allows us to restrict the number of possible inferences very significantly. For problem specification that contain Horn clauses only, selection of at least one negative literal in all non-unit clauses results in *unit-strategies*, where at least one partner in each paramodulation inference is a positive unit clause and hence the number of literals in generated clauses never becomes larger than the number of literals in the longest premise clause. The benefit of selection also extends to problems with general clauses, although in a lesser degree. In both cases, the use of a good literal selection strategy can make a critical difference at least for current clause selection functions. For problems that use only unit clauses selection does not affect the inference process at all.

At the moment, most existing saturating theorem provers use only a simple fixed literal selection strategy or do not use selection at all. As an example, SPASS [WAB+99, WGR96], the best-known superposition-based prover, selects the largest (by number of symbols) negative literal whenever a clause has more than one maximal literal [Wei99]. For E we have implemented a large number of different selection strategies (see Appendix A.2). However, at the moment a given selection scheme is chosen (either by the user or by a heuristic based on clause set features, compare section 5.1) for all clauses generated during the inference process. It is quite possible that a fully dynamic selection of literals can further improve the performance of the prover.

Learning is definitely a possible way for improving literal selection. However, learning for this choice point does require a fairly detailed analysis of the proof process – down to the literal level. This analysis, in turn, requires a very detailed proof protocol, and implies a high use of system resources and a fairly high amount of implementation work. Moreover,

if we consider a proof search that already uses a standard literal selection function (most of which select exactly one literal whenever this is possible), we again can only get examples of positive search decisions from a single proof protocol. Only proof searches which use weak or no literal selection can give us information about good and bad decisions.

Finally, a learning algorithm for literal selection needs to decide for each literal whether to select it or not, while at the same time fulfilling the constraints of the calculus. If we assume that a single clause is sufficient for an informed decision about literal selection, we still need to give a judgment for each individual literal, significantly complicating the problem. There are classes of problems in which literal selection significantly increases the difficulty of finding a proof[6]. Thus, recovery from a single misclassification can be very difficult.

While good literal selection can improve the performance of a theorem prover, the influence of this choice point nevertheless is limited in scope. Even optimal literal selection will at most decrease the branching factor at each choice point by a small factor. It will not reduce the number of clauses necessary for a proof – in fact, it may well increase this number. And particularly for proof problems where a large part of the search is dominated by unit-equational clauses, literal selection has very limited influence.

Finally, we intend to employ the E system in a combined system, where E implements the bottom-up part of the METOP calculus [Mos96]. METOP does not allow literal selection, and we expect unit inferences to play a particularly important role in the combined proof process.

For these reasons, we have delegated this choice point to future work, and will concentrate on clause selection at the moment.

4.3 Clause Selection and Conventional Evaluation Functions

If we assume all choice points except for the selection of the *given clause* to be fixed, we can remap the proof search onto our general model of a search process:

- A single search state now corresponds to a pair of clause sets, i.e. $M = (2^{Clauses(F,P,V)} \times 2^{Clauses(F,P,V)})$, where the first set contains the processed clauses and the second set contains the unprocessed clauses.
- The transition relation changes as well. The possible transitions correspond to the selection of a clause from the set of unprocessed clauses and the changes caused by a single traversal of the main loop of the algorithm in Figure 4.2.
- As before, there is a large number of possible cost measures, and again the most relevant one is the CPU cost of a certain transition in a given implementation. However,

[6] A very simple example is PLA002-2 from the TPTP 2.1.0, which is trivial without selection, but becomes unsolvable (even for very large resource bounds) with nearly any literal selection scheme we have implemented. This is quite typical for many problems in the PLA (*planning*) domain of TPTP.

a fairly good estimation of the search effort is the number of new clauses generated by a single choice.

- A goal state is a state where the empty clause is contained in the set of unprocessed clauses[7], i.e.
  $G = \{(C, U) \in (2^{Clauses(F,P,V) \times Clauses(F,P,V)} | \square \in U\}$.

- Finally, the start state is $(\emptyset, U_0)$, where U_0 contains the clauses of the original problem specification.

The only open choice point now is the selection of the next clause to process. This selection of the given clause is usually controlled by one or more *heuristic evaluation functions*. A heuristic evaluation function usually maps a clause to a numerical evaluation. However, many evaluation functions take the context of the proof process into account. Therefore we use a more complex definition:

Definition 4.4 (Clause evaluation function)
Let $(E, >_E)$ be a totally ordered set and let $\mathcal{D}_{\overline{SP}}$ be the set of all finite **SP** derivations for a given proof problem. A *clause evaluation function* is a function $eval : Clauses(F, P, V) \times \mathcal{D}_{\overline{SP}} \rightarrow E$.

◀

In most existing theorem provers, the evaluation is an integer number or, more rarely, a real number. Given such an evaluation function, the selection of the next clause to process is typically implemented as shown in Fig 4.5.

The two most obviously fair search strategies for a saturating theorem prover are *level saturation* and *first-in first-out*. In the case of level saturation, clauses are selected according to a *proof level*. Clauses from the original problem specification are assigned level 0. The level of a newly generated clause is the maximum level of its parents increased by one. If contracting inferences are at all taken into account (practical uses of level saturation predate most proof calculi with simplification rules), a clause modified by a contracting inference inherits the level of the main premise. The level-saturation strategy selects clauses with a lower level before clauses with a higher level. The effect is a breadth-first search of the space of all derivable clauses.

Pure level-saturation is fairly hard to describe in terms of an evaluation function as it needs exact information about the parents of a clause. The *first-in first-out* or *FIFO* strategy behaves very similar to level-saturation, but is more specific. In the FIFO case, new clauses are processed in the same order in which they are generated. In terms of an evaluation function, we can describe this strategy as follows:

[7]This slightly simplified definition requires that `select_best()` will always select the empty clause if it is an element of `U`. Note that the empty clause can never be in the set of processed clauses, since the empty clause will neither be inserted into the set nor derived by the interreduction procedure (which will never change the maximal term in a clause and hence cannot eliminate all literals).

Variables and Subroutines:

`S`	Set of clauses
`c`	The selected clause
`H`	Encoding of the search derivation
`eval(c,H)`	Clause evaluation function

```
procedure select_best(S,H)
{
  e := min_{>_E} {eval(c,H) | c∈S};
  select arbitrary c from {c∈S | eval(c)=e};
  return c;
}
```

In practice, the clause weight is typically computed once (after the clause has been created and normalized) and cached.

Figure 4.5: Selection of the given clause

Definition 4.5 (First-in first-out evaluation)
The function $\mathit{FIFOWeight} : \mathit{Clauses}(F, P, V) \times \mathcal{D}_{\overline{SP}} \rightarrow \mathbf{N}^\infty$ is defined by

$$\mathit{FIFOWeight}(\mathcal{C}, N_0 \vdash_{\mathbf{SP}} \ldots \vdash_{\mathbf{SP}} N_n) = \begin{cases} \mathit{min}\{i|\mathcal{C} \in N_i\} & \text{if } \mathit{min}\{i|\mathcal{C} \in N_i\} \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

◀

Both level-saturation and first-in first are very weak search strategies. As only the history, but not the structure of the clause are used for the search decision, very large clauses can be selected very early. This leads to a very early explosion of the search space. Table 4.3 compares the overall performance of E with FIFO and other clause selection heuristics. Tables 4.4, 4.4 and 4.6 show the branching factor of the search space for some examples as a function of time and processed clauses. It is obvious that pure FIFO performs very badly. Most current saturating theorem provers use such history-based strategies only to a very small degree. Instead, they select clauses mainly based on syntactic properties of the clauses themselves.

The most frequently encountered search heuristic, and one of the most successful ones, is based on counting symbols and preferring clauses with a small number of symbols, or a small *clause weight*.

Time limit	5s	10s	50s	100s	200s	300s
FIFO	861	900	948	965	990	1002
Weight$_1$ ($w_f = 1, w_v = 1$)	1128	1175	1280	1322	1349	1373
Weight$_2$ ($w_f = 2, w_v = 1$)	1155	1217	1307	1346	1363	1383
Weight$_3$ ($w_f = 1, w_v = 2$)	1012	1054	1140	1159	1181	1199
RWeight	1169	1218	1307	1345	1382	1406
RWeight/FIFO	1294	1359	1480	1519	1540	1565

Remarks: Shown is the number of successes within the given time limit for all clause normal form problems from TPTP 2.1.0 on a SUN Ultra-60/300. *Weight* entries use pure clause weight with the given values of w_f and w_v. *RWeight* uses $w_f = 2, w_v = 1$ and multiplies the weight of maximal terms and the weight of maximal literals with the additional factor $f_{max} = 1.5$. The last entry combines the same *RWeight* strategy and *FIFO* with a pick-given ratio of 5 to 1. We used the standard term ordering and selection of the largest negative literal.

Table 4.3: Comparative performance of search heuristics

Definition 4.6 (Term weight, Term depth, Clause weight)

- Consider $t \in Term(F, V)$. The weight of t with respect to $w_f, w_v \in \mathbf{R}$ is defined as
  - $Weight(w_f, w_v, x) = w_v$ if $x \in V$
  - $Weight(w_f, w_v, f(t_1, \ldots, t_n) = w_f + \sum_{i=1}^{n} Weight(w_f, w_v, t_i)$ otherwise
- The depth of a term t is defined as follows:
  - $Depth(x) = 1$ if $x \in V$
  - $Depth(f(t_1, \ldots t_n)) = 1 + maxDepth(\{t_1, \ldots, t_n\}) \cup \{0\})$
- The weight of a clause $\mathcal{C} = s_1 \simeq t_1 \vee \cdots \vee s_n \simeq t_n$ (with respect to $w_f, w_v \in \mathbf{R}$) is defined by

$$CWeight(w_f, w_v, \mathcal{C}) = \sum_{i=1}^{n} (Weight(s_i) + Weight(t_i))$$

◀

Most current saturating theorem provers use clause weight or variations of it as their main search control heuristic. The most common way of *tuning* a theorem prover for a given domain or problems involves selecting values w_f and w_v for the clause weight heuristic. Typical values are $w_f = 1$, $w_v = 1$ or $w_f = 2$, $w_v = 1$. Table 4.3 shows that clause weight heuristics perform much better than FIFO. It also shows the significant differences between different instances of the clause weight heuristic. Tables 4.4 to 4.6 make this difference even more obvious, but also show that very similar strategies behave very different on the different problems.

There are various reasons for the success of the very simple weight-based approach:

Processed clauses	10	20	50	100	500	1000	2000
INVCOM							
FIFO	24	67	324	1298	-	-	-
Weight_1	9	-	-	-	-	-	-
Weight_2	17	4	-	-	-	-	-
Weight_3	9	-	-	-	-	-	-
RWeight	8	-	-	-	-	-	-
RWeight/FIFO	10	-	-	-	-	-	-
BOO007-2							
FIFO	7	80	913	3663	92090	N/A	N/A
Weight_1	7	34	60	104	126	2660	6922
Weight_2	7	39	57	194	1120	2638	3467
Weight_3	7	22	37	127	1231	6620	11284
RWeight	7	28	57	168	1934	2956	7606
RWeight/FIFO	8	22	117	350	6201	21202	27591
LUSK6							
FIFO	12	134	825	2474	76749	N/A	N/A
Weight_1	13	21	72	108	2293	7219	16416
Weight_2	13	21	106	87	2324	3267	8799
Weight_3	13	21	72	108	2342	3597	5138
RWeight	13	21	73	163	1891	7033	12449
RWeight/FIFO	13	32	99	300	4140	12840	67759

Remarks: Shown is the number of remaining unprocessed clauses after a given number of clauses has been processed. A dash implies that the proof has been found before that number of clauses has been processed, a N/A entry that the number of clauses could not be processed with a limit of 128 MB in less than 300 seconds. Note that our prover automatically removes descendants of clauses recognized as composite, i.e. the number of actually generated clauses typically is much higher than the number of unprocessed clauses. Experimental setup and heuristics are as described in Figure 4.3.

Table 4.4: Branching of the search space over processed clauses

- Small clauses are typically more general than larger clauses, i.e. they represent knowledge about more situations in a more compact form than larger clauses. In practice, this means that they can very often be used in contracting inferences to simplify other clauses or even to show there redundancy.

- Smaller clauses usually have fewer potential inference positions. Thus, processing smaller clauses is more efficient. It is also likely to yield relatively few new clauses, and to yield relatively small clauses, thus restricting the explosion in the search space.

Processed clauses	10	20	50	100	500	1000	2000
HEN011-3							
FIFO	10	22	163	341	4920	39128	203109
Weight_1	6	3	38	61	349	1212	2892
Weight_2	6	3	38	66	377	1192	5005
Weight_3	6	3	37	74	196	554	1477
RWeight	6	9	41	57	234	737	1197
RWeight/FIFO	6	3	56	133	767	3320	9908
PUZ031-1							
FIFO	20	20	5	10	-	-	-
Weight_1	18	13	5	10	-	-	-
Weight_2	18	13	5	19	-	-	-
Weight_3	18	13	5	10	-	-	-
RWeight	18	13	5	14	-	-	-
RWeight/FIFO	18	13	9	-	-	-	-
SET103-6							
FIFO	83	76	106	166	8282	36579	122038
Weight_1	84	76	95	174	1069	3866	9471
Weight_2	84	76	91	218	1673	3378	15033
Weight_3	83	79	110	159	516	2806	5620
RWeight	84	76	103	179	1610	4175	38323
RWeight/FIFO	84	75	90	205	1990	12002	-

Remarks: See previous table.

Table 4.5: Branching of the search space over processed clauses (continued)

- Finally, it is the aim of saturating proof proof procedures to produce the empty clause and hence to make the unsatisfiability of a clause set explicit. Clauses with fewer literals, and hence of lower weight, are more likely to degenerate into the empty clause by appropriate contracting inferences.

Pure symbol counting results in fairly powerful strategies. However, there is a variety of modifications that can further improve this heuristic.

The first variation is to assign different base weights to different function symbols. This is implemented in DISCOUNT. While this can dramatically improve the performance of the prover for some problems, it is usually hard to select good weights. There are some approaches to automate this task for a single domain using learning techniques, see Section 6.1.

Another variation is to weight individual terms and literals in a clause in different ways. One approach is to use the term ordering to determine which parts of a clause to select. DISCOUNT implements the *GTWeight* strategy that only considers the maximal term(s)

(with respect to the used reduction ordering) in each (unit) clause. It also implements an (potentially incomplete) strategy that always prefers orientable unit clauses over unorientable. As generalizations of these early heuristics, E realizes a weight function that allows arbitrary multipliers for the weight of maximal terms within each literal and maximal literals within each clause. E also allows different weight multipliers for positive and negative literals. For further details see Appendix A.2.1.

Table 4.3 shows that the strategy that gives a relatively high weight to maximal terms outperform all of the traditional clause weight approaches, although the evaluation process is more expensive in terms of CPU time. There are two reasons for this success. First, in the ordering-constraint calculi like completion and superposition, only maximal terms are used for generating inferences. Therefore the number of possible inferences is determined by maximal terms only. Non-maximal terms influence the size of newly generated clauses, but (usually) not their number. Secondly, for the case of unit-clauses, orientable clauses, i.e. clauses with exactly one literal, can always be used as rewrite rules. Unorientable equations, on the other hand, require an expensive ordering comparison for each attempt, and in many cases cannot be used for simplification at all. A disadvantage of ordering-based heuristics, on the other hand, is the relatively high computing cost necessary determine maximal terms and literals. Normally this is only necessary for the tiny percentage of processed clauses.

A very different approach is taken by *goal-directed* search heuristics. As we described in Section 2.5, a formula typically consists of two parts: The specification of an algebraic structure (which is satisfiable, i.e. has at least one model) and a (negated) *goal* or *query*. As the specification is satisfiable, all proofs for the problem have to involve the goal[8]. Goal-directed heuristics make use of this feature and attempt to select clauses that are likely to be applicable to reduce the goal to the empty clause.

Goal-directed heuristics are very hard to implement for the general case. Many theorem provers only read an unstructured set of clauses. Even if the theorem prover does distinguish between specification clauses and goal clauses in the input, the set of goal clauses (i.e. the set of all clauses derived using at least one goal or goal-derived clause) grows very fast for most problems. Therefore, there is no small and fixed set of goals to use as a target for the heuristic.

However, there is an important special case in which goal-directed heuristics can be used more easily. Proof procedures for unit-equational problems that are based on Knuth-Bendix-completion [KB70, HR87, BDP89] typically only have to deal with a single ground goal. For this case, a couple of goal-directed heuristics have been realized in DISCOUNT. These include heuristics that prefer unit clauses where one term can match or unify with a subterm of the goal and heuristics that prefer clauses that are structurally similar to the goal. For details see [DF94].

[8]This property is used by non-equational saturating theorem provers in the *set-of-support* strategy [WRC65], and by analytic theorem provers to limit the set of *start clauses* [Lov68, Lov78]. Neither strategy can easily be applied to equational reasoning.

So far, goal-oriented heuristics are mostly useful in special domains or for some few selected examples. However, they can be very useful in combination with other strategies e.g. in the TEAMWORK approach (see below).

For modern high-performance theorem provers, a single clause selection heuristic is insufficient. They typically combine two or more such heuristics. The first well-known implementation of such an approach is the *pick-given ratio* in Otter (see [McC94]). Otter allows the alternating selection of clauses according to a weight-based evaluation function and according to the FIFO-strategy. The *pick-given ratio* describes how many clauses shall be selected according to which criterion. Typically, Otter selects 4 out of every five clauses according to weight and one according to age. This concept has been copied in various other theorem provers. Waldmeister and Vampire are examples for very successful provers that include such a strategy.

One of the advantage of the combination of weight and age based heuristics is that it will usually find short proofs even if relatively large clauses are involved. It will also ensure that all initial axioms are used relatively early. This is particularly effective if the goal (or any other clause necessary for the proof) is large compared to the other input clauses. Table 4.5 shows that for two of the three non-unit problems in our standard test set this effect can reduce the number of clauses that need to be processed before a proof is found, and the results in table 4.6 demonstrate the same effect if we consider proof times and not number of processed clauses. Table 4.3 finally shows that a strategy interleaving clause weight based heuristics (modified by an ordering) with FIFO is much stronger than any of the individual heuristics, at least over the examples from the TPTP problem library.

A similar interleaving heuristic is used in DISCOUNT for the case that the goal contains variables and hence *narrowing* has to be applied to the goal. In DISCOUNT, new goals generated by narrowing are called *critical goals*, and the prover can be set to process critical goals and critical pairs (ordinary unit clauses derived during completion) in an arbitrary ratio.

E has extended these concepts and allows the combination of an arbitrary number of heuristics, where each heuristic additionally can concentrate on a certain class of clauses (goals, non-goals, ground clause, etc). The complete method of specifying composite search heuristics is described in Appendix A.2.1.

The composite heuristics described above combine strategies in a relatively fine-grained way. However, search strategies and heuristics can also be combined in a much more coarse way. We have already described the *startover* strategy implemented in Gandalf and p-SETHEO in Section 4.1. A more complex way to combine different strategies is TEAMWORK [Den93, AD93, ADF95, Den95, DK96, DKS97], a knowledge-based distribution concept for certain search processes that has been implemented in DISCOUNT.

A system that uses the TEAMWORK method has four types of components: experts, specialists, referees and a supervisor. *Experts* and *specialists* are the components actively working on the solution of a given problem. In the case of DISCOUNT most of them are (equational) theorem provers. They work independently for given periods of time, using their own method or their own view on the problem. All experts employ the same basic proving technique (unfailing completion). They only differ in the way they select facts for

processing. Specialists, on the other hand, may employ any correct means to generate new equations, and can also support the supervisor in administrative tasks.

After the experts and specialists have worked for a set period, a *team meeting* takes place. In the first phase of a team meeting the work of all active experts and specialists is judged by *referees*. A referee has two tasks: Measuring the overall progress of an expert or specialist in the last period (resulting in a *measure of success*), and selecting outstanding new results (unit clauses). The results of the referees (measures of success and outstanding results) are collected by the *supervisor*. The supervisor determines the most successful expert and uses its complete search state as a base for a new working period. It also incorporates the outstanding results of the other experts and specialists into this state. The supervisor then determines the composition of the team for the next working period and broadcasts the new search state to all experts and specialists.

TEAMWORK has been fairly successful, but suffers from the fact that at least the currently existing implementation requires a fairly homogeneous cluster of workstations, and is very sensitive to small differences in the performance of these machines. Such disturbances are hard to avoid in a multi-user environment, however, the negative impact can be limited if using DISCOUNT's goal-directed heuristics.

4.4 Summary

In this chapter we have developed a proof procedure for superposition-based theorem proving. We have discussed the search problem resulting from this algorithm and have identified the relevant choice points. We also gave a sufficient criterion for the refutational completeness of the proof procedure.

Using experimental data and pragmatic arguments, we have offered solutions for most of these choice points. We have identified the selection of the next clause to process as the most important class of decisions made during the proof search and have determined this choice point to particularly suitable for the use of learning techniques.

We also identified literal selection and the selection of a good term ordering as potential candidate sfor future work using both meta-knowledge and proof intrinsic knowledge for literal selection and pure meta-knowledge for the selection of term orderings.

Finally, we conducted a survey on the conventional methods used for clause selection in existing theorem provers.

Runtime in seconds	1	5	10	20	50	100	200	300
BOO007-2								
FIFO	6380	17676	19165	37138	81746	145288	N/A	N/A
Weight_1	2447	5166	7466	20458	-	-	-	-
Weight_2	2812	6685	15164	3763	-	-	-	-
Weight_3	1668	6794	12266	13152	7801	24768	14645	71255
RWeight	3007	8475	-	-	-	-	-	-
RW/FIFO	5899	16569	29229	-	-	-	-	-
LUSK6								
FIFO	4889	11847	20416	33170	57314	98168	154478	199160
Weight_1	5233	13572	5890	21987	-	-	-	-
Weight_2	4145	9460	7445	-	-	-	-	-
Weight_3	3653	9221	14362	24352	-	-	-	-
RWeight	1603	7907	-	-	-	-	-	-
RW/FIFO	5172	11739	19599	30938	61894	110220	193770	N/A
HEN011-3								
FIFO	3377	8256	14666	21302	50352	106322	180441	236153
Weight_1	772	1681	2438	3674	-	-	-	-
Weight_2	600	2362	4378	7438	23097	-	-	-
Weight_3	428	926	1424	2404	1677	-	-	-
RWeight	639	1760	4264	8676	29670	62232	-	-
RW/FIFO	1944	4918	8989	13846	-	-	-	-
SET103-6								
FIFO	7486	22184	39034	67136	160079	N/A	N/A	N/A
Weight_1	2409	4788	6772	-	-	-	-	-
Weight_2	2564	6850	13450	30598	71523	-	-	-
Weight_3	728	3532	4708	6637	13713	19023	26919	32745
RWeight	3468	10662	37988	61519	166534	264446	N/A	N/A
RW/FIFO	5211	-	-	-	-	-	-	-

Remarks: Shown is the number of remaining unprocessed clauses after a given time limit. A dash implies that the proof time is lower than the time limit, a N/A entry that the number of clauses could not be processed with a limit of 128 MB. The occasional strong reduction in the number of clauses seen e.g. for the LUSK6 example in the entries for 5 and 10 seconds with Weight_1 is an example for the effect of descendent removal already discussed for the previous table. Experimental setup and heuristics are again as described in Table 4.3. The INVCOM and PUZ031-1 examples are proved in less than a second regardless of heuristic and are omitted from the table.

Table 4.6: Branching of the search space over time

Chapter 5

Representing Search Control Knowledge

In this chapter we introduce data structures for representing knowledge about proof problems and proof searches. We also describe how to extract a relatively compact representation of important search decisions during a proof search from actual protocols of the inferences a prover performed during proof searches.

In automated theorem proving, the main objects we deal with on the inference level are terms, clauses and sets of clauses. However, most existing learning algorithms, especially those that are able to cope with approximate knowledge and contradictory data, work on fixed-length vectors of numerical values. We introduce *numerical features* for (sets of) terms, clauses, and related structures in Section 5.1. Numerical features can be used to represent these objects in a form that allows traditional machine learning algorithms to operate on them. In particular, there exist strong and efficiently computable *distance measures* for feature vectors. These distance measures can be used to induce a notion of *similarity* between clause sets, and hence between proof problems.

These advantages come at a price, however. Numerical features necessarily abstract from most of the properties of recursive structures, and hence limit what kind of knowledge can be expressed. We therefore will use numerical features only for representing proof problems, and rely on learning algorithms that works directly on terms for learning clause evaluations. In order to get a uniform interface for this algorithm, we encode more complex structures, like equations and clauses, as terms. Furthermore, to abstract from arbitrary choices made by the user, we generalize these terms into *term patterns*. Term and pattern representations for clauses are described in Section 5.2.

Section 5.3 finally introduces a representation for search decisions made during a proof search. We describe how to transform a protocol of a proof search into a (relatively small) set of *annotated term patterns* that represent the relevant part of a proof search and the search decisions taken.

5.1 Numerical Features

One way to represent the typical data structures occurring in automated theorem proving is by abstracting their properties into a finite set of numerical (or boolean) *features*.

Definition 5.1 (Term features)

- A function $f : \mathit{Term}(F,V) \to \mathbf{R}$ is called a *term feature function* or simply *term feature* and the value $f(t)$ for a term $t \in \mathit{Term}(F,V)$ is called a *feature value* of t.
- If $f(\mathit{Term}(F,V)) = \{0,1\}$, we call f a *Boolean feature*.
- Let $f_1,\ldots,f_n$ be features. Then the function $\overline{f} : \mathit{Term}(F,V) \to \mathbf{R}^n$ defined by $\overline{f}(t) = (f_1(t),\ldots,f_n(t))$ is a *feature vector function* and $\overline{f}(t) = (r_1,\ldots,r_n) \in \mathbf{R}^n$ is called a *feature vector*.

◀

Typical term features used in theorem proving are e.g. the number of variable occurrences in a term, the number of different variables in a term, the *term weight* (see Definition 4.6), or the depth of a term.

The concept of features can easily be extended to clauses and even sets of clauses:

Definition 5.2 (Clause features, Clause set features)

- A function $f : \mathit{Clause}(F,P,V) \to \mathbf{R}$ is called a *simple clause feature function* or *clause feature* and the value $f(\mathcal{C})$ for a clause $\mathcal{C}$ is called a *(clause) feature value*.
- A function $f : 2^{\mathit{Clause}(F,P,V)} \to \mathbf{R}$ is called a *clause set feature function* or *clause set feature* and the value $f(\{\mathcal{C}_1,\ldots,\mathcal{C}_n\})$ is called a *feature value* for the set of clauses $f(\{\mathcal{C}_1,\ldots,\mathcal{C}_n\})$.
- As before, if a clause feature or clause set feature only maps onto the values 0 and 1, we call it a *boolean feature*.

Feature vectors for clauses and clause sets are defined analogous to feature vectors for terms.

◀

Typical features for clauses are e.g. the number of literals, the number of symbols, the clause weight or the clause depth. Finite clause sets are typically described by features like number of clauses in the set, number of function symbols of a given arity, average number of literals, or average clause weight. Such features are usually selected in an ad-hoc manner based on the experience of system developers, and are refined by experimental evaluation. Table 5.1 shows a list of some term and clause features described in the literature, table 5.2 shows some clause set features.

As finite length feature vectors are very accessible to traditional AI approaches like symbolic machine learning algorithms and neural networks, they have been used for controlling search decisions in both learning and hand-optimized theorem provers.

Feature	Sources
Number of literals in a clause	[CL73, SE90, SE91, Gol91, Gol94]
Number of negative literals in a clause	[SE90, SE91, Gol91, Gol94]
Number of distinct predicate symbols	[CL73, SE90, SE91, Gol91, Gol94]
Number of occurrences of constant function symbols	[SE90, SE91, Gol91, Gol94, Fuc96, Fuc97b]
Number of distinct function symbols	[SE90, SE91, Gol91, Gol94, Fuc96, Fuc97b]
Number of variable occurrences	[SE90, SE91, Gol91, Gol94, Fuc96, Fuc97b]
Depth of a term or clause	[CL73, Fuc96, Fuc97b]
Weight of a term or a clause	[Fuc96, Fuc97b]

Table 5.1: Term and clause features

Feature	Sources
Number of clauses	[Fuc96, Fuc97b, SB99]
Are all clauses unit?	
Are all clauses Horn?	
Are there variables in negative clauses?	
Are there non-constant function symbols in any clauses?	
Number of function symbols of a given arity	[Fuc97a, SB99]
Average term depth of terms occurring in the set	[SB99]

Remarks: Features without reference are implemented in locally used theorem provers (SETHEO, E-SETHEO, p-SETHEO, E) and have not yet been described in publications. We are aware from personal communications that many of them are used in other theorem provers as well. These aspects, however, are rarely published.

Table 5.2: Clause set features

One of the first approaches to learning heuristic evaluation functions was *least square estimation* (see [CL73], pp.154ff and [SF71]), applied to linear polynomials of the feature vector components. This work, however, seems to have been of little influence.

In [SE90, SE91] and similarly in [Gol91, Gol94] the authors use numerical features to describe *connection tableaux* (which can be seen as terms over an extended signature), representing partial proof attempts of the theorem prover SETHEO. The resulting feature vectors are used as input for a multi-layer perceptron (i.e. a neural network for supervised learning) to learn heuristic evaluation functions.

[Fuc96, Fuc97b] describes another use of features for learning search control heuristics for DISCOUNT. Each feature of a (unit-equational) clause is associated with a set of *permissible values* determined by analyzing a successful proof attempt. New clauses are

evaluated by summing the minimal distances (modified by a weight coefficient for each feature) of their feature values with a permissible value for this feature.

A very common use of features is the definition of *distance measures* (or, dual to this, of *similarity measures*). This allows the application of *case-based reasoning* (see e.g. [Kol92] for an overview).

Definition 5.3 (Absolute distance measures)
Assume $a, b \in \mathbf{R}^n$, $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$.

- $dist_M(a, b) = \sum_{i=1}^{n} |a_i - b_i|$ is called the *Manhattan distance* between a and b.
- $dist_E(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$ is called the *Euclidean distance* between a and b.
- Let $w = (w_1, \ldots, w_n) \in \mathbf{R}^n$ be a vector or *weights*. The *weighted Euclidean distance* between a and b (for the weight vector w) is $dist_W(a, b) = \sqrt{\sum_{i=1}^{n} (w_i(a_i - b_i))^2}$.

◀

Sometimes it is necessary to combine features with very different value ranges, where each feature nevertheless has about the same importance. In order to still allow each feature to contribute to the same degree, we need to *normalize* either the distances or the features. Normalizing the feature values (by taking the average or the maximum value of the feature over all occurring objects as a normalizing factor) has a serious disadvantage: It requires a-priori knowledge of all feature values. In particular, if we add a new object with new feature values, we need to recompute all normalized feature values. Moreover, this may change the distance between two otherwise unaffected objects (and may even change the differences in distance between to vectors).

Example: Consider the feature vectors $a = (1, 1)$, $b = (1, 0.5)$ and $c = (0.6, 1)$. If we normalize the feature values using the maximum feature values, a and b remain unchanged. The Manhattan distances between a and the two other vectors are $dist_M(a, b) = 0.5$ and $dist_M(a, c) = 0.4$, i.e. c is closer to a as b. However, if we consider a fourth vector, $d = (1, 2)$, this changes. We get the normalized vectors $a' = (1, 0.5)$, $b' = (1, 0.25)$ and $c' = (0.6, 0.5)$. Now $dist_M(a', b') = 0.25$ and $dist_M(a', c') = 0.4$, and b' is closer to a' than c'.

Similar effects occur with other distance measures and other global normalization schemes.

To avoid these undesirable effects, we now introduce relative distances for feature values and distance functions based on them.

Definition 5.4 (Relative distance measures)
We define a *relative difference function* $\delta : \mathbf{R} \times \mathbf{R} \to [0; 1]$ by

$$\delta(a, b) = \begin{cases} 0 & \text{if } a = 0 \text{ and } b = 0 \\ \frac{a-b}{2 \times max(|a|, |b|)} & \text{otherwise} \end{cases}$$

Now assume $a, b \in \mathbf{R}^n$, $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ as in the previous definition

- $rdist_M(a, b) = \sum_{i=1}^{n} |\delta(a_i, b_i)|$ is called the *relative Manhattan distance* between a and b.
- $rdist_E(a, b) = \sqrt{\sum_{i=1}^{n} \delta(a_i, b_i)^2}$ is called the *relative Euclidean distance* between a and b.
- Let $w = (w_1, \ldots, w_n) \in \mathbf{R}^n$ be a vector of *weights*. The *weighted relative Euclidean distance* between a and b (for the weight vector w) is given by $rdist_W(a, b) = \sqrt{\sum_{i=1}^{n} (w_i \times \delta(a_i, b_i))^2}$.
- If $rdist : \mathbf{R}^n \to \mathbf{R}$ is a relative distance measure, we call $\overline{rdist} : \mathbf{R}^n \to [0; 1]$ defined by $\overline{rdist}(a, b) = \frac{rdist(a,b)}{n}$ for all $a, b \in \mathbf{R}^n$ the corresponding *normalized relative distance measure*.

◀

Feature-based distance functions are e.g. used in the approach described by [Fuc97a], where a suitable search guiding function for a new proof is selected by comparing the performance of the functions on the *nearest neighbour* (the problem with the smallest distance for some distance measure) from a data base containing the feature vectors and the performance of a finite set of strategies for a set of proof problems. This approach used the simple Euclidean distance.

Many other theorem provers use features to select one of multiple strategies as well. All recent sequential versions of SETHEO [MIL+97] use boolean features (e.g. presence of non-Horn clauses or presence of true function symbols) to select the search strategy. The 1998 version of p-SETHEO [Wol98a] used a large number of Boolean and numerical features to select a set of search strategies to run in parallel. The (conventional) automatic mode of our own prover, E [Sch99b], as used in the CASC-16 ATP competition, uses a set of 8 Boolean and ternary features to select one of about 20 different proof strategies. Similarly, the new, combined system E-SETHEO [SW99, WL99, Wol98b, Wol99b] uses a vector of Boolean features to determine which of a list of strategy schedules to use on a given problem.

We have based the selection of training examples for our learning system on clause set features. A preliminary version of this selection mechanism, using the relative Manhattan distance, was successfully implemented for the *DISCOUNT/TSM* system and has been described in [SB99]. We have further generalized this approach for full clausal logic by modifying the set of features used. For details see Section 7.2.

5.2 Term and Clause Patterns

The algebraic structure described by a given specification is independent of the actual set of function and variable symbols used. Moreover, certain relevant properties of terms and

clauses in a proof search may also be independent from the function symbols used. If we want to transfer knowledge between different but similar problems, and if we want to capture general, signature-independent syntactic features of terms and clauses, we need to cope with this situation. There are a number of different approaches to deal with the problem, many of which require the substitution of function symbols with different ones.

A *function symbol renaming* is a function that does exactly this:

Definition 5.5 (Function symbol renaming, Symbol renaming)
Let $sig = (F, ar)$ be a signature and V be a set of variables. A *function symbol renaming* τ is a (not necessarily injective) function $\tau : F \to F$ with $ar(\tau(f)) = ar(f)$ for all $f \in F$.

It is extended to a function $\tau : Term(F, V) \to Term(F, V)$ in the usual way, i.e.

- $\tau(x) = x$ for $x \in V$.
- $\tau(f(t_1, \ldots, t_n)) = \tau(f)(\tau(t_1), \ldots, \tau(t_n))$

We write $\tau = \{f_1 \leftarrow g_1, \ldots, f_n \leftarrow g_n\}$ to denote a function symbol renaming τ with $\tau(f_1) = g_1, \ldots, \tau(f_n) = g_n$, and we denote the set of all function symbol renamings for a signature *sig* with $fsr(sig)$.

If $\mu = \tau \circ \sigma$ with $\tau \in fsr(sig)$ and $\sigma \in \Sigma_{perm}(V)$, we call μ a *symbol renaming.*

◀

Function symbol renamings are a very general concept, and can be used to describe more specific techniques used in learning for theorem proving. The most often used approach is trying to map symbols from the signature of a given specification to the signature used in the representation of certain piece of learned knowledge.

Definition 5.6 (Signature match)
Let $sig_1 = (F_1, ar_1)$ and $sig_2 = (F_2, ar_2)$ be two signatures. A signature match τ from sig_1 to sig_2 is a function symbol renaming $\tau \in fsr(sig_1 \cup sig_2)$ with $\tau(f) \in F_2$ for all $f \in F_1$ and $\tau(f) = f$ for all $f \in F_2$.

◀

Signature matching is used e.g. in DISCOUNT to detect applicable knowledge for *learning by pattern memorization* [Sch95, DS96a, DS98], to select one of many proof control plans [DK96] and to decide on the selection of *focus facts*for *flexible reenactment* [FF97, Fuc97b, Fuc96].

While signature matching has been used in learning, it does have some disadvantages. In particular, signature matches are not unique, but the number of possible matches rises combinatorially with the number of symbols of each arity. Even if we try to use function symbol renamings to match term structures onto each other, matches are still not unique:

Example: Consider $sig_1 = \{f/2, a/0, b/0\}$ and $sig_2 = \{g/2, c/0, d/0\}$. If we want to map $f(a, b)$ onto $g(c, d)$, we can use four different function symbol renamings, two of which are injective:

$$\tau_1 = \{f \leftarrow g, a \leftarrow c, b \leftarrow c\}$$
$$\tau_2 = \{f \leftarrow g, a \leftarrow c, b \leftarrow d\}$$
$$\tau_3 = \{f \leftarrow g, a \leftarrow d, b \leftarrow c\}$$
$$\tau_4 = \{f \leftarrow g, a \leftarrow d, b \leftarrow d\}$$

This ambiguity leads to limited scalability even for systems using *transformational analogy*, i.e. systems like PLAGIATOR [KW94, KW96] (which often use even stronger versions of *second order matching*), although these systems only have to find a single match to solve the proof problem at hand. In our approach, where we attempt to extract knowledge from a large number of source problems for a single new problem, and moreover want to evaluate a large number of individual clauses efficiently, this approach is unacceptable.

Instead, we compute a unique representation for all terms (and later equations and clauses) of a certain structure, at the cost of losing inter-clause relationships between function symbols. That is, we rename some (user-defined) function symbols on a per-clause basis in such a way that the resulting term-representation of each clause becomes minimal in a total ordering on equivalent representations.

Definition 5.7 (Lexicographical term ordering)
Let F be a set of function symbols with associated arities, and let V be a set of variable symbols. Let further $\succsim$ be a quasi-ordering total up to $\approx$ on $F \cup V$. Let $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ be two terms from $Term(F, V)$ (remember that variable symbols are syntactically equivalent to constants here). Then the lexicographical extension of $\succsim$ to terms is recursively defined as follows:

$$s \succsim_{tlex} t \text{ if } \quad f \succ g \text{ or}$$
$$f \approx g \text{ and } (s_1, \ldots, s_n) \succ_{tlex_{lex}} g(t_1, \ldots, t_m)$$

◀

Theorem 5.1 (Totality of $\succ_{tlex}$)

- The relation $\succ_{tlex}$ for a total precedence $\succ$ on function symbols and variables is a total ordering on terms.
- The relation $\succsim_{tlex}$ for a given precedence ordering $\succsim$ total up to $\approx$ is a quasi-ordering on terms. The equivalence part of $\succsim_{tlex}$ is given by $[f(t_1, \ldots, t_n)]_{\approx_{tlex}} = \{g(t'_1, \ldots, t'_n) \mid g \approx f, t'_i \in [t)i]_{\approx_{tlex}} \text{ for all } i \in \{1, \ldots, n\}\}$.

Proof: The ordering is equivalent to the normal lexicographical ordering on the flat word representation of terms, for which the result is well known.

■

With this ordering, we can now define *representative patterns* for terms ([Sch95, DS96a, DS98] define a slightly simpler form of representative patterns, [SB99] introduces patterns

equivalent to the ones used here, but using a less general framework). The basic idea is that we split the set of function symbols into two parts, a part F_f which we consider to play a special, well-defined role in a term (usually function symbols introduced by our encodings), and a set of (usually user-defined) function symbols F_g that we want to abstract from. Function symbols from F_g are *generalized* (replaced by new symbols playing the role of limited second order variables), function symbols from F_f remain fixed.

Definition 5.8 (Representative term patterns)
Let $F = F_f \uplus F_g$ be a set of function symbols and (F, ar) be a signature. Let $\mathcal{S} = \biguplus_{i \in \mathbf{N}} S_i$ with $S_i = \{f_{ij} | j \in \mathbf{N}\}$ be an enumerable set composed of (mutually disjoint) enumerable sets of new symbols. We define $ar_S : S \to \mathbf{N}$ by $ar_S(f_{ij}) = i$ and $sig = ((F, ar) \cup (S, ar_S))$. Finally, let V be a set of variable symbols and let $\succ$ be a a total precedence on $S \cup F \cup V$ with the following properties:

1. $f_{ij} \succ f_{i'j'}$ iff $i > i'$ or $i = i'$ and $j > j'$ for all $f_{ij}, f_{i'j'} \in S$
2. $f \succ f'$ for all $f \in S$ and $f' \in F$
3. $f \succ x$ for all $f \in S \cup S$ and $x \in V$

Then we define the following terms:

- $Term(S \cup F, V)$ is called the set of *term patterns* for $Term(F, V)$.
- The term $s \in Term(F \cup S, V)$ is called a *term pattern* for $t \in Term(F \cup S, V)$, if there exists a *pattern substitution* $\mu = \sigma \circ \tau$ with $\mu(s) = t$, where $\sigma \in \Sigma_{perm}(V)$ and $\tau \in fsr(sig)$ with $\tau(f) = f$ for all $f \in F$.
- A term s is called *more general* than t if s is a term pattern for t, but not vice versa. If s is a pattern for t and t is a pattern for s, s and t are called *equivalent patterns*.
- A term $s \in Term(F \cup S, V)$ is called *most general pattern* for a term $t \in Term(F, V)$ *with respect to* F_f, if there exists a pattern substitution μ with $\mu(f) = f$ for all $f \in F_f$ and there is no more general pattern with this property for t. We denote the set of most general patterns with respect to F_f for t with $mgp_{F_f}(t)$. If we speak only of *the most general pattern* for a term, we assume the case $F_f = \{\}$.
- The *representative term pattern (with respect to F_f)* for a term t is the term pattern $s = min_{\succ_{tlex}} mgp_{F_f}(t)$. As the following theorem states, the representative term pattern for a term is unique, therefore we can write $repgen(s, F_f)$ to denote the representative term patterns for s with respect to F_f. Again, if we omit F_f we assume $F_f = \{\}$.

◀

Theorem 5.2 (Uniqueness of the representative term pattern)
The representative term pattern for a term s with respect to set of function symbols F_f and a given precedence $\succ$ is unique.

Proof: The precedence $\succ$ is total, hence, by Theorem 5.1, $\succ_{tlex}$ is total. As the representative term pattern is defined as the minimum of a set of patterns with respect to this ordering, it is well-defined.

■

The representative pattern for a term can be computed very efficiently by traversing the term once and substituting function symbols from F_g in their order of appearance with suitable new symbols from S. To illustrate this point, we give some simple examples of terms and their representative patterns.

Example: Assume $sig = \{f/2, g/1, h/1\}$.

- Consider the term $t_1 \equiv f(g(x_2), g(x_1))$.
  - We first substitute f with the new $f_{21} \in \mathcal{S}$ (the minimal new symbol with arity 2).
  - The next symbol encountered is g, which is substituted with f_{11}.
  - We then continue to normalize the variables. The resulting pattern is $repgen(t_1, \{\}) = f_{21}(f_{11}(x_1), f_{11}(x_2))$.
  - Similarly, $repgen(t_1, \{f\}) = f(f_{11}(x_1), f11(x_2))$.
- Consider the term $t_2 \equiv f(g(h(x_2)), g(x_1))$.
  - $repgen(t_2, \{f, g, h\}) = f(g(h(x_1)), g(x_2))$.
  - $repgen(t_2, \{\}) = f_{21}(f_{11}(f_{12}(x_1)), f_{11}(x_2))$.
  - $repgen(t_2, \{g\}) = f_{21}(g(f_{11}(x_1)), g(x_2))$.

Literals and clauses can be easily encoded as terms. For equations and inequations, there are few obvious variations. We choose to encode both in an equivalent way:

Definition 5.9 (Term encoding of equations and literals)
Let $sig = (F, ar)$ be a signature. We extend sig by adding two new symbols: $sig' = sig \uplus \{eq/2, neq/2\}$.

- Let $s \simeq t$ be an equation over $Term(F, V)$. Then the term $eq(s, t)$ is a *term encoding* of $s \simeq t$. Keep in mind that we consider equations to be symmetric, i.e. there are two term encodings for each (non-trivial) equation.
- Similarly, let $s \not\simeq t$ be a negated equation. Then the term $neq(s, t)$ is a term encoding of $s \not\simeq t$.
- We denote the set of term encodings for a literal $s \simeq t$ by $T_{enc}(s \simeq t)$.

◀

For clauses, there are two obvious possibilities. On the one hand, we can consider a clause with n literals as a term with n principal arguments, on the other hand, we can treat a clause as a *list* of literals and encode it as such.

Definition 5.10 (Term encodings for clauses)
Let $sig = (F, ar)$ be a signature and let $\mathcal{C} \equiv \mathcal{L}_1 \vee \ldots \vee \mathcal{L}_n$ be a clause.

- Let $sig_1 = sig \uplus \{eq/2, neq/2, or_0/0, or_1/1, \ldots\}$ be an extension of sig. Then any term $or_n(L'_1, \ldots, L'_n)$ with $L'_i \in T_{enc}(\mathcal{L}_i)$ for all $i \in \{1, \ldots, n\}$ is a *flat term encoding* of $\mathcal{C}$. We denote the set of all flat term encodings for a clause $\mathcal{C}$ by $T_{flat}(\mathcal{C})$.
- Let $sig_2 = sig \uplus \{eq/2, neq/2, or/2, nil/0\}$ be another extension of sig. Then the set of *recursive term encodings* of $\mathcal{C}$, $T_{rec}(\mathcal{C})$ is defined inductively:
  - $T_{rec}(\square) = nil$.
  - $T_{rec}(\mathcal{L} \vee \mathcal{R}) = \{or(L, R) \mid L \in T_{enc}(\mathcal{L}), R \in T_{rec}(\mathcal{R})\}$.

◀

Keep in mind that the order of literal encodings in the term encoding of a clause is indeterminate, as the original clause is a (unsorted) multi-set of literals.

Example: Consider the clause $\mathcal{C} = g(g(x)) \simeq x \vee f(g(x)) \not\simeq g(f(x))$.

- A flat clause encoding of $\mathcal{C}$ is the term $or_2(eq(g(g(x)), x), neq(f(g(x)), g(f(x)))$.
- An alternative flat clause encoding is the term $or_2(neq(g(f(x)), f(fg(x)), eq(g(g(x)), x))$.
- A recursive clause encoding of the same clause is $or(neq(g(f(x)), f(fg(x)), or(eq(g(g(x)), x)), nil))$.

Both term encodings for clauses have interesting properties with respect to term-based learning algorithms:

- Flat term encoding immediately groups a clause with clauses of the same length. The length of a clause is an important feature: Positive unit clauses are used as rewrite rules, cutting back on the search space, and after all, we have found a proof if we encounter a clause of length 0. Therefore, having this feature encoded in an easily accessible way may help in the classification of clauses. Flat term encodings also treat all literals as equivalent as far as many learning algorithms are concerned: All literals of a clause are at the same depth level, and will usually have the same influence for a given evaluation function.

- On the other hand, the recursive term encoding may map initial parts of clauses of different length together. This allows the generalization from clauses of a certain length to those of bigger length. This is much more difficult with the flat term encoding. Moreover, this representation better reflects the fact that a clause can appear as a substructure of a larger clause.

We can require the representative pattern of a clause (encoded as a term) to be minimal with respect to $\succ_{tlex}$ as we did with ordinary terms. However, equations are symmetric and thus have two equivalent term representations. Similarly, clauses are defined as multi-sets, and thus the number of equivalent term representations for a clause rises super-exponentially with the number of literals: A clause with n equational literals has $n!2^n$ different but equivalent syntactic representations. Thus, computing the representative pattern for a clause would become very expensive at least for a straightforward implementation. We can avoid much of this cost (for the average case) if we pre-order terms and literals with respect to some ordering that is stable under function symbol renaming, i.e. an ordering that only compares the syntactic *structure* of two terms.

Definition 5.11 (Stable under symbol renaming)
Let $\succsim$ be a quasi-ordering on $Term(F, V)$. It is called *stable under symbol renaming* iff

1. $s \succ t$ implies $\mu(s) \succ \mu(t)$ for all $s, t \in Term(F, V)$ and all symbol renamings μ.
2. $s \approx t$ implies $\mu(s) \approx \mu(t)$ for all $s, t \in Term(F, V)$ and all symbol renamings μ.

◀

Obviously, if a term is a pattern for another term, both are equivalent in the equivalence part of any quasi-ordering that is stable under symbol renaming.

There is a variety of quasi-orderings that are stable under symbol renamings. Among these are orderings induced by term weights (which are independent of the actual function symbols), orderings taking only topological features of the term into account, and combinations of both.

As we want to use such an quasi-ordering to pre-order literals and clauses for pattern computation, there are some points to consider.

- The quasi-ordering should be as strong as possible, i.e. the equivalence part should be rather small and the strict part should be large. This minimizes backtracking due to choices between equivalent possibilities.
- Large terms and literals should be selected early. As each renamed function symbol limits the possible choices for later terms, this again serves to minimize backtracking and search.
- Finally, the ordering should be efficient to compute.

We will now define some quasi-orderings that are stable under symbol renaming and lead to a particular ordering that fulfills the above criteria.

Definition 5.12 (Some stable quasi-orderings on terms)
Consider a set of terms $Term(F, V)$ over a signature (F, ar).

- Assume $w_f, w_v \in \mathbb{R}$. Then $\succsim_{W(w_f,w_v)}$ is defined by
  - $s \approx_{W(w_f,w_v)} t$ if $Weight(s, w_f, w_v) = Weight(t, w_f, w_v)$ for all $s, t \in Term(F, V)$.
  - $s \succ_{W(w_f,w_v)} t$ if $Weight(s, w_f, w_v) > Weight(t, w_f, w_v)$ for all $s, t \in Term(F, V)$.
- Let $\succsim_{ar}$ be the precedence on function symbols and variables defined by $f \approx_{ar} g$ if $ar(f) = ar(g)$ and $f \succ_{ar} g$ if $ar(f) > ar(g)$ for all $f, g \in F \cup V$. Then $\succsim_{ar_{tlex}}$ is a quasi-ordering stable under symbol renaming.
- Finally, we define $>_{preord}$. Consider two terms $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv g(t_1, \ldots, t_m)$ (where variables are once more treated as function symbols of arity 0).

$$
\begin{array}{ll}
s \geq_{preord} t \text{ if} & s \succ_{W(-2,-1)} t \text{ or} \\
& s \approx_{W(-2,-1)} t \text{ and } g \succ_{ar} f \text{ or} \\
& s \approx_{W(-2,-1)} t \text{ and } f \approx_{ar} g \text{ and} \\
& (s_1, \ldots, s_n) \geq_{preord_{lex}} (t_1, \ldots, t_n)
\end{array}
$$

◀

In the above definition of $\geq_{preord}$, the first condition ensures that terms with a high function symbol count are smaller than terms which only contain a few symbols. The second condition, comparing the arities of function symbols, speeds up the comparison by (sometimes) removing the need for recursive descent. The final condition adds strength to the ordering by breaking ties, and ensures that if we compare terms that differ only in the order of their arguments, terms which put arguments with a large number of function symbols first are smaller than those ordered in any other way.

Using this ordering, we can finally define representative patterns for clauses.

Definition 5.13 (Representative clause patterns)
Let $F = F_f \uplus F_g$ be a set of function symbols, let $sig = (F, ar)$ be a signature with $\top/0 \in F_f$ and let V be a set of variables.

- Let $sig_1 = sig \uplus \{eq/2, neq/2, or_0/0, or_1/1, \ldots\}$ be an extension of sig. Assume S and $\succ$ (on $F' = F \cup \{eq, neq, or_0, or_1, \ldots\}$) as in Definition 5.8. We define $\succ_{cf}$ by $s \succ_{cf} t$ if $s >_{preord} t$ or $s \approx_{preord} t$ and $s \succ_{tlex} t$ for all $s, t \in Term(F', V)$. Then the *flat representative pattern (with respect to F_f)* for a clause $\mathcal{C}$ is the term $min_{\succ_{cf}}\{repgen(c, F' \backslash F_g) \mid c \in T_{flat}(\mathcal{C})\}$.
- Let $sig_2 = sig \uplus \{eq/2, neq/2, or/2, nil/0\}$ be another extension of sig. Assume S and $\succ$ (on $F'' = F \cup \{eq, neq, or, nil\}$) as in Definition 5.8. We define $\succ_{cl}$ by $s \succ_{cl} t$ if $s >_{preord} t$ or $s \approx_{preord} t$ and $s \succ_{tlex} t$ for all $s, t \in Term(F'', V)$. Then the *recursive representative pattern (with respect to F_f)* for a clause $\mathcal{C}$ is the term $min_{\succ_{cl}}\{repgen(c, F'' \backslash F_g) \mid c \in T_{rec}(\mathcal{C})\}$.

◀

As with representative term patterns, representative patterns for clauses are unique:

Theorem 5.3 (Uniqueness of representative clause patterns)
The flat representative clause pattern and the recursive representative clause pattern for a clause $\mathcal{C}$ and with respect to a set F_f of fixed symbols are unique. We write $repclause_{flat}(\mathcal{C})$ and $repclause_{rec}(\mathcal{C})$, respectively.

Proof: The same argument as for Theorem 5.2 holds.

■

To compute the representative clause pattern, we interleave the construction of the term representation and the building of the symbol renaming. We start with an empty list L of literals, an empty symbol renaming and a set M containing all literals of a clause. At each stage, we compute the set of potentially $>_{cf}$ or $>_{cl}$ minimal term encodings for literals in M. For each such alternative E, we have to explore a different possibility. We append the term encoding to L, remove the corresponding literal from M, and continue the symbol renaming to cover the symbols in E. The procedure is applied recursively until all literals have been removed from M. The final lists L correspond to possible term patterns for the clause, the representative clause pattern is the $>_{cf}$ or $>_{cl}$ minimal of these patterns.

Example: Again consider the clause $\mathcal{C} = g(g(x)) \simeq x \vee f(g(x)) \not\simeq g(f(x))$. Then any term representation of the second literal is smaller in $>_{preord}$ than any term representation of the first literal, as the second literal has a higher function symbol count than the first one. Hence, any minimal clause pattern has to start with an encoding of the first literal.

There are two different possibilities to encode this literal, $neq(f(g(x)), g(f(x)))$ and $neq(g(f(x)), f(g(x)))$. Let us consider the first one. It leads to the function symbol renaming $\{f \leftarrow f_{11}, g \leftarrow f_{21}\}$. For the second literal, the minimal encoding is obvious, and we get the flat clause pattern

$$or_2(neq(f_{11}(f_{12}(x)), f_{12}(f_{11}(x))), eq(f_{21}(f21(x)), x))$$

If we consider the second option, we get the function symbol renaming $\{g \leftarrow f_{11}, f \leftarrow f_{21}\}$, and we get the pattern

$$or_2(neq(f_{11}(f_{12}(x)), f_{12}(f_{11}(x))), eq(f_{11}(f11(x)), x))$$

This pattern is smaller than the first one (due to the lexicographical comparison of f_{12} and f_{11} in the encoding of the second literal) and is in fact the representative flat clause pattern for $\mathcal{C}$.

In practice, the different choices at each stage are explored using a standard backtracking algorithm. Due to the strong pre-ordering of terms and literals, the average case behaviour of this algorithm is good enough for our application, although the worst case behaviour is still exponential.

5.3 Proof Representation and Example Generation

We will now describe how to represent the important decisions during a successful proof search by a relatively small number of *annotated clauses*. The core idea is to select clauses that actually contribute to a proof and clauses that can be derived from those in at most a few inference steps. To achieve this, we represent the proof derivation as a graph and analyze the relationships encoded in this graph.

In Chapter 4 we represented a proof search as a sequence of paths in the graph whose nodes correspond to derivable clause sets and whose edges correspond to inference steps. For the analysis of a given proof search, it is more useful to represent this as a graph whose nodes are labeled with the individual clauses appearing during the proof search and whose edges describe the inferences used to generate each clause.

If we consider the inference system **SP**, we can distinguish between different kinds of inference rules. The most important distinction is between generating and contracting inferences. However, the second class of inferences can be partitioned into 2 subclasses: Modifying inferences and deleting inferences.

Definition 5.14 (Inference types, Premise types)
Consider the inference system **SP** from page 24.

- The inference rules (ER), (SN), (SP) and (EF) are called *generating inference rules* and inferences resulting from their application are called *generating inferences*.
- The inference rules (RN),(RP),(SR),(DD) and (DR) are called *modifying inference rules* and inferences resulting from their application are called *modifying inferences*. In these rules, we call the rightmost premise in the rule the *main premise* and the other premises (if they exist) *side premises*. We call the rightmost clause in the conclusion the *main conclusion*.
- Finally, the inference rules (CS), (ES) and (TD) are called *deleting inference rules* (resulting in deleting inferences). We again call the rightmost premise the *main premise* and the other premises (if they exist) *side premises*.

◀

We can now define the *proof derivation graph* corresponding to a given proof derivation. In the graph, we distinguish between different kinds of edges: Edges that represent the transition from a clause to a modified clause (quoting edges), edges connect premises and

conclusion of a generating inference, edges connect side premises with the main conclusion in modifying inferences, and edges expressing the relationship between subsumed and subsuming clauses.

Definition 5.15 (Proof derivation graph)
Let $D = S_0 \vdash S_1 \vdash S_2 \vdash \ldots S_n$ be a finite proof derivation with $S_0 = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$. We assume that all clauses occurring in D are distinct objects, even if they have the same form.

- The *graph representation* of D is defined as the graph $G_n = (N_n, Q_n \cup G_n \cup S_n \cup M_n)$ resulting from the following recursive construction:
  - $G_0 = (\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}, \emptyset)$.
  - Assume that $G_i = (N_i, Q_i \cup G_i \cup S_i \cup M_i)$.
    - Assume that $S_i \vdash S_{i+1}$ with a generating inference with premises $\mathcal{C}'_1, \ldots, \mathcal{C}'_l$ and conclusion $\mathcal{C}$. Then $N_{i+1} = N_i \cup \{\mathcal{C}\}$, $Q_{i+1} = Q_i$, $G_{i+1} = G_i \cup \{(\mathcal{C}'_1, \mathcal{C}), \ldots, (\mathcal{C}'_l, \mathcal{C})\}$, $S_{i+1} = S_i$, and $M_{i+1} = M_i$.
    - Assume that $S_i \vdash S_{i+1}$ with an application of (RN), (RP) or (SR), with main premise $\mathcal{C}'_1$, side premise $\mathcal{C}'_2$ and main conclusion $\mathcal{C}$. Then $N_{i+1} = N_i \cup \{\mathcal{C}\}$, $Q_{i+1} = Q_i \cup \{(\mathcal{C}'_1, \mathcal{C})\}$, $G_{i+1} = G_i$, $S_{i+1} = S_i$, and $M_{i+1} = M_i \cup \{(\mathcal{C}'_2, \mathcal{C})\}$.
    - Assume that $S_i \vdash S_{i+1}$ with an application of (DD) or (DR) with premise $\mathcal{C}'$ and conclusion $\mathcal{C}$. Then $N_{i+1} = N_i \cup \{\mathcal{C}\}$, $Q_{i+1} = Q_i \cup \{(\mathcal{C}', \mathcal{C})\}$, $G_{i+1} = G_i$, $S_{i+1} = S_i$, and $M_{i+1} = M_i$.
    - Assume that $S_i \vdash S_{i+1}$ with an application of (CS) or (ES) with main premise $\mathcal{C}'_1$ and side premise $\mathcal{C}'_2$. Then $N_{i+1} = N_i$, $Q_{i+1} = Q_i$, $G_{i+1} = G_i$, $S_{i+1} = S_i \cup \{(\mathcal{C}'_2, \mathcal{C}'_1)\}$, and $M_{i+1} = M_i$.
    - Finally, if $S_i \vdash S_{i+1}$ with an application of (TD), then $G_{i+1} = G_i$.
- Edges in Q_n are called *quote-edges*, edges in G_n are called *generating edges*, edges in S_n are called *subsumption edges* and edges in M_n are called *modifying edges*.
- A *clause family* in G_n is a set of clauses connected by quote-edges.
- A proof derivation graph is called successful, if it contains the empty clause $\Box$.

◀

It is possible to extend this construction to infinite derivations. In practice, however, any derivation will stop after a finite time, either due to finding a proof, i.e. deriving the empty clause, due to saturating the clause set without finding a proof (and thus proving it to be satisfiable), or due to lack of resources.

The concept of a *clause family* in the proof derivation graph corresponds to the persistence of a clause in the theorem proving process, where a clause can be repeatedly modified during processing, but is considered as the same object. That means, a clause family contains all representations a single clause object takes during the proof process.

Example: Consider the following partial inference protocol describing a proof for the INVCOM problem. The first column shows a running number, the second column describes the inference and the third column shows the (main) conclusion of the inference. We use the running number to refer to the corresponding clause.

```
 0 axiom              f(X,0) = X.
 1 axiom              f(X,i(X)) = 0.
 2 axiom              f(f(X,Y),Z) = f(X,f(Y,Z)).
 3 axiom              f(a,i(a)) != f(i(a),a).
 4 (RN) with 1 on 3   0 = f(i(a),a)
 5 (SP) with 2 and 0  f(X,f(Y,0)) = f(X,Y).
 6 (RP) with 0 on 5   f(X,Y) = f(X,Y).
 7 (SP) with 2 and 0  f(X,f(0,Y)) = f(X,Y).
 8 (SP) with 2 and 1  f(X,f(Y,i(f(X,Y)))) = 0.
 9 (SP) with 2 and 1  f(X,f(i(X),Y)) = f(0,Y).
10 (SP) with 2 and 2  f(f(X,Y),f(Z,U)) = f(f(X,f(Y,Z)),U).
11 (RP) with 2 on 10  f(X,f(Y,f(Z,U))) = f(f(X,f(Y,Z)),U).
12 (RP) with 2 on 11  f(X,f(Y,f(Z,U))) = f(X,f(f(Y,Z),U)).
13 (RP) with 2 on 12  f(X,f(Y,f(Z,U))) = f(X,f(Y,f(Z,U))).
14 (TD) with 13
   .
   .
   .
```

Figure 5.1 shows the resulting proof graph. Non-trivial clause families are $\{5, 6\}$ and $\{10, 11, 12, 13\}$.

If a proof derivation graph represents a successful proof search, it will contain the empty clause, and we can denote a subgraph of it as a proof object. Moreover, we can classify clauses and clause families according to their *distance* from the proof.

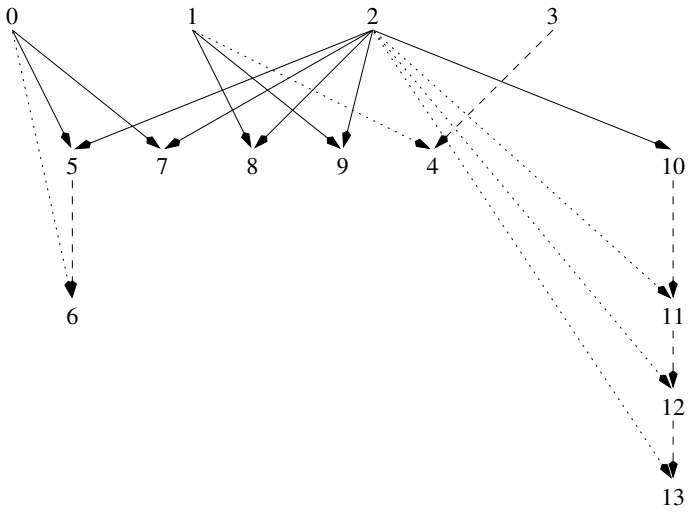
Definition 5.16 (Proof path, Proof object, Proof distance)
Let $G = (N, E)$ be a successful proof derivation graph for some proof problem $\mathcal{F}$.

- Any path of the form $\mathcal{C}, \ldots, \square$ with $\mathcal{C} \in \mathcal{F}$ is called a *proof path*.
- The subgraph $G' = (N', E')$ with $N' = \{n \in N | n \text{ is on a proof path } \}$ and $E' = \{(k, k') \in E | (k, k') \text{is part of a proof path}\})$ is called a *proof object* or simply *proof*.
- The *proof distance* of a clause in G is defined as follows:

$$pd(\mathcal{C}) = \begin{cases} 0 & \text{if } \mathcal{C} \text{ is on a proof path} \\ 1 + max(pd(pred(\mathcal{C}))) & \text{otherwise} \end{cases}$$

- The *proof distance* of a clause family in G is the minimum proof distance of any clause in the family.

◀



Remarks: Solid lines show generating edges, dashed lines show quote-edges, dotted lines show modifying edges. There are no subsumption edges in the example.

Figure 5.1: Example proof derivation graph

5.3.1 Selecting representative clauses

Now let us assume that the proof derivation is generated by a *given-clause* algorithm as depicted in Figure 2.35. We can distinguish two cases for a clause family: Either one of its elements becomes the given clause, or none of its elements is ever selected for processing. In the second case, the clauses in the family cannot participate in the proof process at all. Therefore we cannot extract any knowledge about their potential value for the proof search from them. This leads us to the following premise:

> **Premise:** Only clauses from clause families that contain at least one selected clause can be used to represent evaluated search decisions.

We can represent the *positive* search decisions as the families of clauses with a proof distance of 0. Each of these families contains at least one clause that contributed to the proof, and hence the decision to select one of the clauses was necessary for finding this particular proof.

However, we may also want to represent *negative* search decisions, i.e. clause selections that did not contribute to the proof. In general, there are a lot more such clause examples – see table 4.2 for numbers. Using *all* clause families with a proof distance greater than 0 as negative examples therefore is impractical, as it would quickly overwhelm the ability of any learning algorithm to sort through all examples.

Let us consider a hypothetical *perfect* search heuristic, i.e. a heuristic that will only select clauses that contribute to a single proof for each proof problem.

There are two ways to achieve this: First, we can assign a very good evaluation to all clauses that are necessary for the proof. However, alternatively we can describe the proof

by *rejecting* all clauses not necessary for the proof, i.e. by assigning a very bad evaluation to these clauses. In the case of a perfect heuristic, we only need to reject all clauses that are directly (in one inference) derivable from the axioms and the contributing clauses to force the theorem prover into a perfect proof derivation.

In practice, we cannot expect such a perfect heuristic. However, it obviously still makes sense to select clauses close to the proof process as examples.

> **Premise:** Search decisions during a proof search can be described by a set of clause families with a low proof distance.

Finally, we need to decide how to represent a clause family. We can of course represent a family by all of its members. This is likely to lead to a very good knowledge transfer even if the proof search for a new problem is slightly different. However, it also leads to a relatively large amount of data (and associated overhead), and leads to the problem of splitting the evaluation of the clause family to the individual members. However, we do not need to use *all* clauses in a clause family as examples of search decisions. As newly generated clauses are typically evaluated exactly once, and modified clauses inherit the evaluation of their main premise, we only need to pick the clauses from a clause family that where actually evaluated during the proof search. While this may lead to a slightly more brittle system, we can solve this problem by combining knowledge about many different proof searches.

> **Premise:** The relevant clauses to describe a search process of the given-clause algorithm are the clauses from clause families with a low proof distance that were evaluated by the algorithm.

Our results presented in Chapter 8 indicate that this subset of clauses (with the annotations described below) contains sufficient information to reproduce proofs and even allows us to generalize to new proof problems.

5.3.2 Assigning clause statistics

We have now selected a set of clauses to describe a proof process. However, the value of a clause for the proof process depends on more information than just the proof distance.

- A clause that contributes to many proofs is more useful than a clause that contributes to fewer or no proofs.
- A clause that simplifies or subsumes many other clauses helps to prune the search space and is thus potentially useful.
- A clause that generates a lot of useless successors is very bad for the search process.

To take these features into account we assign an annotation to each clause selected for representing a proof process. Note that the above features depend on the size of the proof

search – in a proof search that e.g. only processes 20 clauses, a clause can at most subsume 19 other clauses. In a proof search with 20,000 processed clauses, this is very different. To correct for this effect, we set the individual annotation values in relation to the potential number of inferences of the relevant type.

The annotation of a clause consists of a vector of numbers with the following numbers:

- The proof distance *pd* of the clause family the clause was taken from
- The number *mp* of modifying inferences in the proof in which clauses from the clause's family were used in as side clauses, divided by the number of processed clauses
- The number *mn* of other modifying inferences in which clauses from the family were used in as side clauses, divided by the number of generated clauses
- The number *gp* of successors contributing to the proof generated using clauses from the family, divided by the number of processed clauses (note that only processed clauses have a chance to contribute to the proof)
- The number *gm* of superfluous successors generated using clauses from the family, divided by the number of generated clauses
- The number *sc* of clauses subsumed by clauses from the clause family, divided by the number of processed clauses

All of these values can be easily computed by analysis of the edges of the proof derivation graph.

Note that similar values are used in our previous approach [DS96a, DS98] and are also one of the information sources used by the *referees* in TEAMWORK or TECHS [DF99] to select potentially useful facts.

5.4 Summary

In this chapter we have first described the representation of terms, clauses and clause sets by numerical features. We have also described how distance measures on feature vectors can be used to induce a notion of *similarity* between clause sets. In particular, we have introduced *relative distance measures* and *normalized relative distance measures*.

We have then discussed the problem of abstracting from a given signature and introduced term patterns and clause patterns. Clause patterns allow us to represent clauses by a unique term, and in a way that abstracts from an arbitrary subset of function symbols.

Finally, we have described a way to represent the important search decisions during a successful proof search as a set of annotated clauses. Clauses are selected as representative if they participated in the search process and are close to the final proof. Their role in the proof process is described by a vector of numerical values computed by analyzing the proof derivation graph.

Chapter 6

Term Space Maps

In the previous chapter we have represented search decisions as individual annotated clauses. Our aim is to learn good evaluations of new clauses (representing search alternatives) form this representation. To achieve this aim, we first transform clauses into patterns (i.e. into first order *terms* over an extended signature), and compute an evaluation from the annotations at a clause. We now want to transform this set of evaluated terms into an operational form that allows us to use the collected information for the evaluation of new clauses. This task is at the very core of the learning prover, and can be stated as an independent machine learning problem.

Only relatively few machine learning algorithms can deal well with both recursive structures (like terms) and numerical evaluations. In this chapter we will first give a short discussion and survey of term-based learning algorithms. We will then introduce *learning by term space mapping*, a class of new learning algorithms for term classification and evaluation. Term space mapping represents knowledge by partitioning the set of training examples based on a term abstraction, and by storing class and evaluation-relevant information with each partition. New terms are mapped onto the resulting structure and evaluated in different ways according to the data stored in the matching partitions.

6.1 Term-Based Learning Algorithms

Term-based learning algorithms operate on terms and aim at learning either a classification or an evaluation. In the general case, input to a term-based learning algorithm is a set of terms associated with an desired evaluation (which can represent a classification)[1]. This set of examples is called the *training set*. The output of the learning algorithm is a knowledge representation that allows us to compute a likely evaluation for new terms. An algorithm can be evaluated by testing its performance on a second set of terms with known evaluations, the *test set*.

[1]Note that evaluation-based algorithms can be turned into classification algorithms by comparing the evaluation to a limit, and that similarly classification algorithms can return (crude) evaluations by associating each class with a particular value.

For our case, we can list a couple of desirable properties for the learning algorithm:

- We want to acquire heuristic search control knowledge, preferably in the form of a numerical evaluation. The learning algorithm should be able to represent such knowledge.
- There is only some vague lore about which properties of a term (or term-encoded larger structure) are relevant for its evaluation. Therefore the algorithm should not be strongly biased for or against certain properties (or at least have a known and adjustable bias).
- It is well-known that different search strategies perform very different on different problems and problem domains. We would therefore like to be able to train a search strategy for a new problem on a limited set of experiences from similar problems. This *learning on demand* requires fast learning algorithms. At the very least, the time for learning should not be significantly longer than the expected proof search time for hard problems. Given the usage of current theorem provers, this means that learning times should be lower than about one minute.
- Even more important than efficiency during the learning phase is efficiency during application of the knowledge. As (nearly) each clause generated during the proof search will be evaluated against the learned knowledge, this evaluation should be not more expensive in terms of CPU time as other frequent operations on clauses.
- Even among similar proofs and problems, we cannot expect a one-to-one correspondence between terms or clauses useful for all problems. This has two consequences: First, we have to be able to deal with contradictory and approximate information during the learning phase. Clauses useful for one proof problem may be superfluous in others. Secondly, the learned knowledge should be in a form that enables a prover that uses it to recover from occasional miss-classifications or miss-evaluations during the application phase.

Unfortunately, these goals conflict. In particular, stronger learning algorithms capable of learning more complex concepts typically need more examples and/or more training time than simpler methods. We therefore have to strive for a compromise.

Existing learning algorithms used for term evaluation or classification can be placed in a spectrum from purely symbolic to purely numerical algorithms. Fig 6.1 shows this spectrum and qualitatively places the learning algorithms that have been applied to theorem proving.

The most simple purely symbolic learning algorithm is the memorization (or storing) of terms of particular classes. Despite the simplicity of this approach, it has been successfully applied to theorem proving (*Flexible reenactment*, [Fuc96, Fuc97b], relies on the selection and application phases to introduce generalization to new proof examples). Other symbolic learning algorithms include *explanation-based generalization* (EBG) [MKKC86, MCK$^+$89, Wus92, Etz93] (or *-learning*) and *inductive logic programming*[Mug92, MR94]. Explanation-based learning generates descriptions of classes by

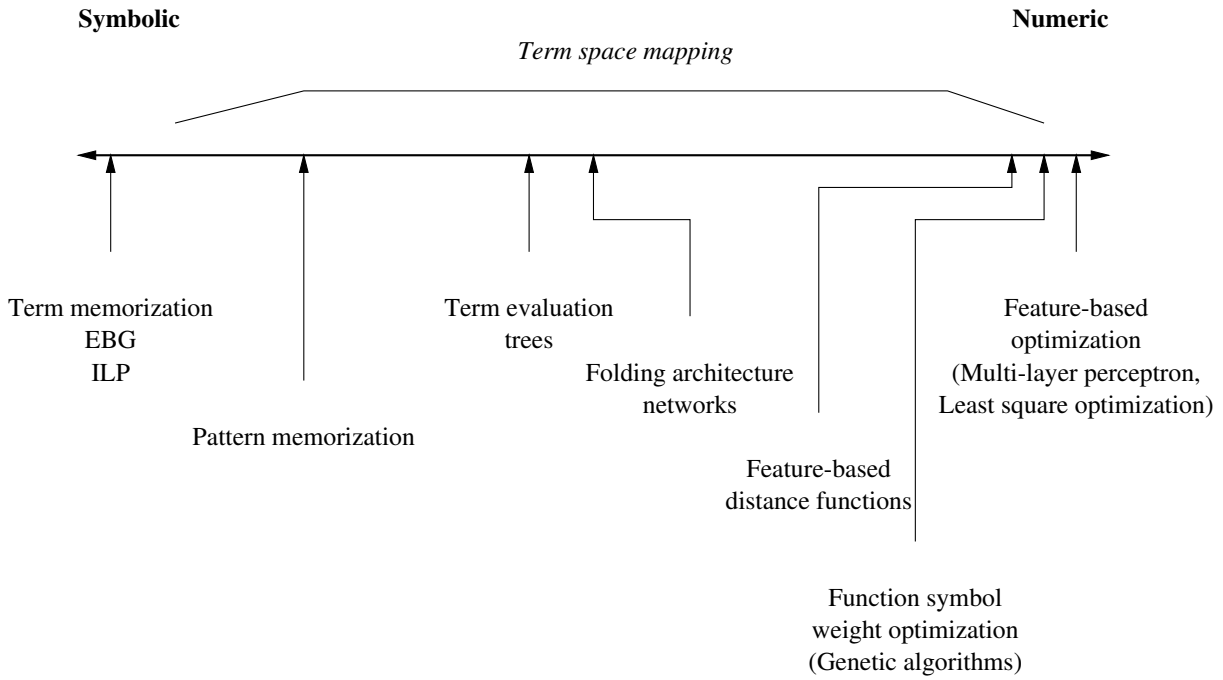


Figure 6.1: The symbolic-numeric spectrum for learning algorithms

justified generalization of derivations in a background theory. As it is based on logical derivations and justified generalization, it necessarily derives valid hypotheses. However, it is unable to derive knowledge not logically implied by training examples and background theory, and merely finds in some way better (more *operational*) descriptions of the classes. Inductive logic programming also makes use of a background theory, but allows the speculation of hypotheses. Both approaches, however, are not very suitable for dealing with the quantitative, approximate and partially inconsistent knowledge typical for search control heuristics, especially as very little knowledge about the domain exists and hence usually no background theory is available.

Most purely numerical learning algorithms on terms work on feature vectors and have already been described in Section 5.1. In this case, the explicit background theory is replaced by the implicit assumptions used in the numerical representation of the symbolic structures. [Fuc95b] describes a slightly different approach: Parameters of a standard weight function (in this case, weights assigned to individual function symbols) for terms are optimized with a genetic algorithm until the resulting heuristic evaluation functions leads to the desired evaluation of clauses.

While such numerical optimization procedures are well-suited for expressing approximate and quantitative knowledge, the transformation of the original term classification problem into a purely numerical problem introduces a very strong bias. Only a limited number of term properties can be encoded naturally, and all other properties of the term are ignored. An additional disadvantage is that most of the more expressive learning al-

gorithms require a large number of training examples and are rather slow, i.e. they cannot be used for *learning on demand* situations.

Hybrid learning algorithms try to avoid some of these disadvantages by combining structure based processing and numerical operations. Primary examples for hybrid learning algorithms are *learning by pattern memorization*, *learning with term evaluation trees* (both described in [Sch95, DS96a, DS98]) and *folding architecture networks* [GK96, Gol99a, KG96, SKG97].

Learning by pattern memorization has been implemented to guide DISCOUNT, a completion-based theorem prover for unit-equational proof problems. The algorithm works by storing representative patterns of equations (which are a special case of representative clause patterns as defined in Definition 5.13) with the desired evaluation. Despite the very limited expressive power of this method, good results have been achieved. In particular, using pattern memorization enabled the prover to prove a wide variety of problems with a single strategy, whereas it previously required a wide range of strategies to achieve the same number of successes. Generalization to new proof problems was also observed, although in a lesser degree.

Term evaluation trees are an early predecessor of the recursive term space maps we will introduce later in this chapter. They work by recursively partitioning a set of terms according to the arity of the top function symbol, and associating an evaluation with each node in the resulting tree. The original implementation lead to improved performance of the DISCOUNT system and allowed the system to proof a number of previously unsolvable problems. However, most successes required the selection of a relatively small set of suitable training examples.

Finally, *folding architecture networks* are probably the most powerful of the existing hybrid learning algorithms. They apply gradient-based training algorithms like *back-propagation through structure* [GK96, KG96] to neural networks that are dynamically unfolded to accommodated the recursive structure of terms. The generic architecture has been proven to be a universal approximator for mappings from directed acyclic graphs to real vector spaces [Ham96, HS97]. Folding architecture networks have been applied to control the proof search of SETHEO [Gol99a, Gol99b] and we have conducted preliminary experiments for applying them for saturating theorem provers [SKG97]. The main disadvantage of folding architecture networks are the long training times (which make *learning on demand* impossible) and a requirement for large numbers of training examples to avoid over-fitting.

Our approach, *term space mapping*, generalizes the concept of term evaluation trees. It also is able to emulate to a certain degree algorithms that assign weights to function symbols and, if applied to patterns, subsumes pattern memorization.

6.2 Term Space Partitioning

As stated above, *term space mapping* describes a class of hybrid learning algorithms. Term space maps partition the set of all terms according to certain criteria and store evaluation-

relevant data with each partition. The partition of the term set is based on one out of a variety of abstractions of terms. While a lot of different abstractions are possible, we will primarily use abstractions that represent a term by its *initial part*, i.e. by the part of the term structure that is relatively close to the top position.

Terms can be represented in various ways as *finite labeled ordered directed acyclic graphs* or *FLODAGS*. Some of these term representations closely reflect the actually implemented data structure in most existing theorem provers. Moreover, they allow the easy definition (and implementation) of useful term abstractions.

Definition 6.1 (Graph representation of terms)
Let $t \in Term(F, V)$ be a term and let M be an arbitrary set. A *graph representation* of t is a labeled ordered graph $((K, E), l)$ with

- $K = \{node(p) | p \in O(t)\}$ for a function $node : O(t) \to M$ with $node(p) = node(q)$ implies $t|_p \equiv t|_q$.
- $E = \{(node(p), node(i.p)) | i.p \in O(t)\}$
- $succ(node(p)) = node(p.1), \ldots, node(p.n)$ with $n = ar(head(t|_p))$
- $l(node(p)) = head(t|_p)$ for all $p \in K$.

The *root node* of a graph representation of a term is $node(\lambda)$. ◀

The most natural representation of a term is a tree, with a one to one mapping of tree nodes and term positions. More exactly, a tree representation of a term is a graph representation with $M = O(t)$ and $node = id$:

Definition 6.2 (Tree representation of terms)
Let $t \in Term(F, V)$ be a term. The *tree representation* of t is the ordered, labeled tree $((K, E), l)$ with

- $K = O(t)$
- $E = \{(p, i.p) | i.p \in O(t)\}$
- $succ(p) = p.1, \ldots, p.n$ with $n = ar(head(t|_p))$
- $l(p) = head(t|_p)$ for all $p \in K$.

◀

Example: Consider the term $t = f(g(a), g(g(a)))$. Then the tree representation for t is given by $T = ((K, E), l)$ with $K = \{\lambda, 1, 1.1, 2, 2.1, 2.1.1\}$ and $E = \{(\lambda, 1), (\lambda, 2), (1, 1.1), (2, 2.1), (2.1, 2.1.1)\}$. The successor nodes are ordered according to the lexicographical extension of the ordering $>$ on $\mathbf{N}$, and the label function is given by $l(\lambda) = f$, $l(1) = l(2) = l(2.1) = g$, $l(1.1) = l(2.1.1) = a$. Figure 6.2a shows the tree.

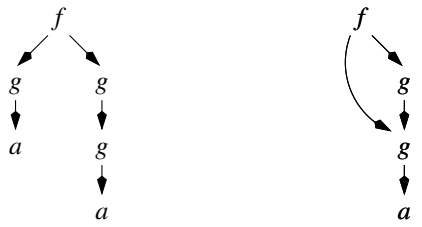


Figure 6.2: Graph representations of $f(g(a), g(g(a)))$

Tree representations are naturally extended to sets of terms (which are mapped onto forests).

As the example shows, the tree representation of a term contains a separate subtree for each subterm, even if subterms appear more than once in the term. Especially for large terms and term sets, it is much more economical to *share* common subterms.

Definition 6.3 (Maximally shared representation of terms)
Let $t \in Term(F, V)$ be a term. The *maximally shared representation* of t is the ordered, labeled directed acyclic graph $((K, E), l)$ with

- $K = \{t|_p | p \in O(t)\}$
- $E = \{(t|_p, t|_{i.p}) | i.p \in O(t)\}$
- $succ(p) = t|_{p.1}, \ldots, t|_{p.n}$ with $n = ar(head(t|_p))$
- $l(s) = head(s)$ for all $s \in K$.

◀

Example: Consider $t = f(g(a), g(g(a)))$ from the previous example. The resulting maximally shared graph is shown in Figure 6.2b.

Using these term representations, we can now easily define top terms:

Definition 6.4 (Top terms)
Let t be a term.

- The *top term* of t at level i, $top(t, i)$, is the term resulting if we replace every node reachable from the root node by a path of length i in the tree representation of t with a fresh variable.

- The *alternate top term* of t at level i, $top'(t, i)$, is the term resulting if we replace every *distinct subterm* at a node reachable from the root node by a path of length i in the tree representation of t with a fresh variable.
- The *compact shared top term* of t at level i, $cstop(t, i)$, is the term resulting if we replace every node reachable from the root node by a path of length i in the maximally shared graph representation of t with a fresh variable.
- The *extended shared top term* of t at level i, $estop(t, i)$, is the term resulting if we replace every node reachable from the root node by a path of length i, but not by any shorter path, in the maximally shared graph representation of t with a fresh variable.

◀

Example: Consider $t \equiv f(g(a), g(g(a)))$ again.

- $top(t, 0) = top'(t, 0) = cstop(t, 0) = estop(t, 0) = x$
- $top(t, 2) = f(g(x), g(y))$
- $top'(t, 2) = f(g(x), g(y))$
- $cstop(t, 2) = f(x, g(x))$
- $estop(t, 2) = f(g(x), g(g(x)))$
- $top(t, 3) = f(g(a), g(g(x)))$
- $top'(t, 3) = f(g(a), g(g(x)))$
- $cstop(t, 3) = f(g(x), g(g(x)))$
- $estop(t, 3) = f(g(a), g(g(a)))$

Now consider $t' \equiv h(a, b, a)$.

- $top(t, 1) = h(x, y, z)$
- $top'(t, 1) = cstop(t, 1) = estop(t, 1) = h(x, y, x)$

Term tops are one example for term abstractions. In general, we allow any term abstraction generated by a function that fulfills rather weak criteria:

Definition 6.5 (Index Functions)
Let $sig = (F, ar)$ be a signature and V be a set of variable symbols. Let M be an arbitrary set (the *index set*).

- A function $I : Term(F, V) \to M$ is called an *index function*, if $I(s) = I(t)$ implies $ar(head(s)) = ar(head(t))$ for all $s, t \in Term(F, V)$.

- It I is an index function with index set M, $ar_I : M \to \mathbf{N}$ is defined by

$$ar_I(m) = \begin{cases} ar(head(t)) & \text{if } \exists t \in Term(F,V),\ I(t) = m \\ 0 & \text{otherwise} \end{cases}$$

◀

Index functions range from very simple functions with few possible index values to the term identity function:

Theorem 6.1 (Some index functions)
Assume *sig* and V as in Definition 6.5. Then the following functions are index functions:

1. $I_{ar} : Term(F,V) \to \mathbf{N}$ with $I_{ar}(t) = ar(head(t))$ for all $t \in Term(F,V)$
2. Assume $i \in \mathbf{N}, i > 0$.
   - $I_{top_i} : Term(F,V) \to Term(F,V)$, $I_{top_i}(t) = top(t,i)$.
   - $I_{top'_i} : Term(F,V) \to Term(F,V)$, $I_{top'_i}(t) = top'(t,i)$.
   - $I_{cstop_i} : Term(F,V) \to Term(F,V)$, $I_{cstop_i}(t) = cstop(t,i)$.
   - $I_{estop_i} : Term(F,V) \to Term(F,V)$, $I_{estop_i}(t) = estop(t,i)$.
3. $I_{symb} : Term(F,V) \to F \cup V$, $I_{symb}(t) = head(t)$.
4. $I_{id} : Term(F,V) \to Term(F,V)$, $I_{id}(t) = t$ (the identity function).

Proof:

1. By definition of I_{ar}, each term is mapped to the arity of its top function symbol.
2. If two terms are mapped to the same index value by any of the functions, they share the top function symbol.
3. I_{symb} is equivalent to I_{top_1}.
4. As for item 2.

■

Note that the term top functions for a depth 0 are not index functions, as they map all terms, regardless of the top function symbol, onto the same new variable.

Under certain circumstances, we can create new index functions out of old ones.

Definition 6.6 (Compatibility of index functions)

- Let $I_1 : Term(F,V) \to M_1$ and $I_2 : Term(F,V) \to M_2$ be two index functions. I_1 and I_2 are called *compatible*, if $ar(head(s)) = ar(head(t))$ for all $s,t \in Term(F,V)$ with $I_1(t) = I_2(s)$.
- A set $I = \{I_1, \ldots, I_n\}$ of index functions is compatible, if all pairs of index functions in I are compatible.

◀

Incompatible index functions can be easily made compatible by modifying their index sets to be disjoint. If index functions are compatible, we can combine them.

Theorem 6.2 (Composite index functions)
Let $I_1 : Term(F,V) \to M_1$ and $I_2 : Term(F,V) \to M_2$ be two compatible index functions and let P be an arbitrary predicate on $Term(F,V)$. Then $I : Term(F,V) \to M_1 \cup M_2$ defined by

$$I(t) = \begin{cases} I_1(t) & \text{if } P(t) \\ I_2(t) & \text{otherwise} \end{cases}$$

is an index function

Proof: Consider two terms s and t with $I(s) = I(t)$. We have to show that $ar(head(s)) = ar(head(t))$. If $P(s)$ and $P(t)$ is equivalent for s and t this follows from the fact that I_1 or I_2 are index functions.

Now let us assume (without loss of generality) that $P(s)$ holds and $P(t)$ does not hold. But then $I_1(s) = I_2(t)$ and therefore $ar(head(s)) = ar(head(t))$ by the definition of compatibility.

■

The last theorem allows us to construct a very large number of different index functions, and to combine index functions representing e.g. specific prior knowledge about good term space partitionings for a given task with more general ones.

6.3 Term Space Mapping with Static Index Functions

We will now describe *term space maps* build around a single predetermined index function. The most general structure used in term space mapping are *basic term space maps*. The three different kinds of term space maps introduced below are instances of this basic data structure. We define *basic term space maps* and *term space alternatives* by mutual recursion:

Definition 6.7 (Basic term space map)
Let $I : Term(F,V) \to M$ be an index function.

- A *term space alternative* (or *TSA*) for the index function I is a tuple $(i, e, (tsm_1, \ldots, tsm_{ar_I(i)}))$ with the following properties:

  1. $i \in I(Term(F, V))$ is an index. We say that the term space alternative is *indexed* by i.
  2. $e \in \mathbf{R}$ is the *evaluation* of the term space alternative.
  3. $tsm_1 \ldots tsm_{ar_I(i)}$ are basic term space maps (not necessarily for I).

- A *basic term space map* (or basic *TSM*) for I is a finite set of *term space alternatives* with the property that each index $i \in M$ indexes at most one alternative.

- The empty term space map is the empty set, written as $\{\}$.

- If $I(t)$ is the index of a term space alternative *tsa* in a basic term space map, we say that the TSM *maps* the term t, and that t *is mapped onto tsa*.

◀

Learning with term space mapping is the construction of term space maps capturing features of particular *training sets*, i.e. (multi-)sets of terms with an associated evaluation. We call such term space maps *representative* for the training set and the index function. Again, the most simple case is the case of a *representative basic term space map*.

Definition 6.8 (Representative basic term space map)
Let M be a multi-set of terms t with associated evaluations $eval(t)$ and let I be an index function. A *representative basic term space map* for I, *eval* and M is a term space map $\{(i, e(i, I, M), (tsm_{i,1} \ldots tsm_{i,ar_I(i)}) | i \in I(M)\}$, where

$$e(i, I, M) = \frac{\sum_{t \in \{t' \in M | I(t') = i\}} eval(t)}{|\{t' \in M | I(t') = i\}|}$$

and where the $tsm_{i,j}$ are arbitrary basic term space maps.

◀

Let us discuss this definition. First, the definition only restricts the top level term space map. It does not require any special properties of the sub-TSM's in the term space alternatives. As such, it defines a whole class of term space maps for each training set and index function. We will restrict this freedom in different ways below. Secondly, each representative basic term space map will split the training set into alternatives, and it will associate the average evaluation of all terms in one alternative with this evaluation. We use this data to evaluate terms when we apply term space maps. In the most simple case we only retrieve a single evaluation for each term.

Definition 6.9 (Flat term evaluation)
Let t be a term, let *tsm* be a basic term space map for some index function I and assume a constant value $e_u \in \mathbf{R}$ (the *evaluation of unmapped terms*). The *flat evaluation of* t *under* *tsm* is defined as follows:

$$fev(tsm, t, e_u) = \begin{cases} e & \text{if there exists a TSA } (i, e, (tsm_1, \ldots, tsm_n)) \text{ with } i = I(t) \\ & \text{in } tsm \\ e_u & \text{otherwise} \end{cases}$$

◀

If we use basic term space maps with flat term evaluation, we only make use of the top-level term space alternatives of a TSM. Consequently, we can use a more simple structure, the *representative flat term space map*.

Definition 6.10 (Representative flat term space maps)
Let again M be a multi-set of terms t with associated evaluations $eval(t)$ and let I be an index function. The *representative flat term space map* for I, *eval* and M is the representative basic term space map

$$rftsm_I(M) = \{(i, e(i, I, M), (\{\}, \ldots, \{\}) | i \in I(M)\}$$

where

$$e(i, I, M) = \frac{\sum_{t \in \{t' \in M | I(t') = i\}} eval(t)}{|\{t' \in M | I(t') = i\}|}$$

as above.

◀

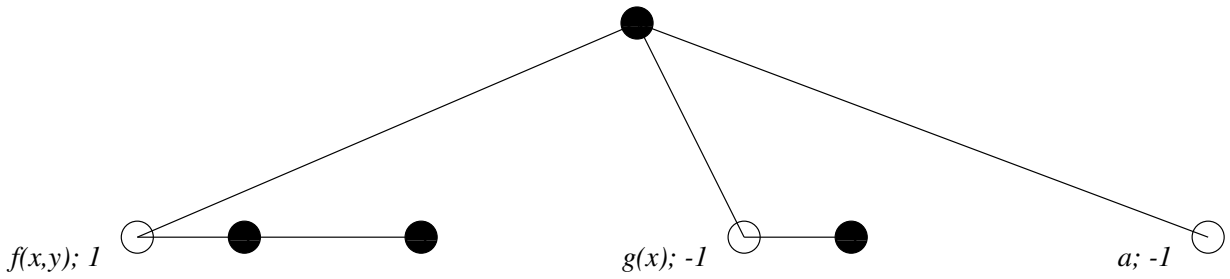
Constructing a representative flat term space map for a training set is straightforward. For an example see page 92.

Flat term space maps evaluate the complete term with respect to a single index function only. Nevertheless, flat term space maps even for the simple index functions described in Theorem 6.1 are sufficiently strong to subsume learning algorithms actually used in theorem provers. In particular, flat term space maps can be used to model *term memorization* and, if applied to representative term or clause patterns, *pattern memorization*. To model term memorization, we have to recognize finite sets of terms. We can use a term space map for classification instead of evaluation by comparing the evaluation to some suitably selected limit:

Definition 6.11 (Term classification with term space maps)
Let $M = M^+ \uplus M^-$ be a set of terms, let *tsm* be a basic term space map and let *tev* be a TSM-based evaluation function (e.g. *fev*, see Definition 6.9). Let $e_u \in \mathbf{R}$ be an evaluation for unmapped term nodes and let $l \in \mathbf{R}$ be the *classification limit*.

We say that *tsm* recognizes a term $t \in M$ with respect to e_u and l if $ev(tsm, t, e_u) > l$ iff $t \in M^+$. We say that *tsm* recognizes M^+ (in M) if it recognizes all terms in M.

◀



Remarks: Solid circles correspond to term space maps, horizontal lines represent sub-TSM sequences and open circles represent term space alternatives. Alternatives are labeled with *index; evaluation*.

Figure 6.3: A representative flat term space map

Let us consider an example for flat term space maps and classification with flat term space maps:

Example: Consider the set $\{f(g(a),b), f(b,b), g(g(a)), a\}$. Terms that contain the symbol f are considered positive examples, i.e. the evaluations are given by $eval(f(g(a),b)) = eval(f(b,b)) = 1$ and $eval(g(g(a))) = eval(a) = -1$. Figure 6.3 shows a graphic representation of the representative flat term space map $rftsm_{I_{top_1}}(M)$.

In this simple example, $fev(rftsm_{I_{top_1}}(M), t, 0) = eval(t)$ for all $t \in M$. Of course all terms in the training set are classified correctly for the classification limit $l = 0$.

The most simple approach to model term memorization is to use the index function I_{id}, to use exactly the set we have to memorize as the training set, to use a constant evaluation for these terms, and to use an evaluation e_u for unmapped terms that differs from this value in the evaluation. However, we can recognize arbitrary finite sets even without using a predetermined value e_u by mapping a larger part of the term space:

Theorem 6.3 (Recognizing finite sets)
Let $M^+ \subseteq Term(F,V)$ be a finite set of terms. Then there exists a training set M of terms with evaluations and an index function I such that $rftsm_I(M)$ recognizes M^+ from $Term(F,V)$ with respect to 0 and 0.

Proof:

Assume a depth limit $d \in \mathbf{N}$, $d = max(Depth(M^+))$, and the training set $M = \{t \in Term(F,V) | Depth(t) \leq (d+2)\}$ with evaluations defined by

$$eval(t) = \begin{cases} 1 & \text{if } t \in M^+ \\ -1 & otherwise \end{cases}$$

Let I be defined by $I(t) = top(t, d+1)$. Then $tsm := rftsm_I(M)$ recognizes M^+:

First note that all terms mapped to a certain alternative have the same evaluation. Hence, the only occurring evaluations are 1 and -1. The limit l for the recognition of terms is 0. Note also that any term t with a depth smaller than $d+1$ has the index t. Now assume an arbitrary term $t \in Term(F, V)$.

- Case 1: $t \in M^+$. Then $I(t) = t$ and hence $fev(tsm, t, 0) = 1$. Ergo t is recognized.
- Case 2: $t \notin M^+, Depth(t) \leq d+1$ Then again $I(t) = t$ and thus $fev(tsm, t, 0) = -1$. Ergo t is recognized.
- Case 3: $t \notin M^+, Depth(t) > d+1$. As M contains *all* terms of depth up to $Depth(t) + 2$, there exists a term $t' \in M$ with $I(t) = I(t')$ and hence $fev(tsm, t', 0) = fev(tsm, t, 0)$. Since $Depth(t) > d$, $t' \notin M^+$ and $fev(tsm, t', 0) = -1$. Again t is recognized.

■

This theorem can be further strengthened. We can use flat representative term space maps to learn all classes that can be described by finite conjunctions and disjunctions of statements about function symbols or subterms at certain term positions, i.e. of statements of the form $t \in M$ *if* $p \in O(t)$ *and* $t_p = t'$. The above proof carries over (using the sum of the length of the longest position p plus the depth of the deepest term t in the class description as a limit), but becomes rather lengthy and requires significant additional terminology. As we are less interested in the theoretical power of term space maps and more in their value to guide theorem proving, we omit it.

As flat term space maps with simple index functions only can take features of a finite initial part of the term into account, they cannot recognize infinite term classes defined by position-independent properties, e.g. the class $\{t \in Term(F, V) \mid \exists p \in O(t), t|_p \equiv a\}$ (if the signature contains at least $\{a/0, b/0, g/1\}$). If we allow additional index functions, such classes can be learned. However, such index functions typically introduce a fairly strong bias. As an extreme example, consider the index function $I : Term(F, V) \to \mathbf{N} \times \{0, 1\}$ defined by

$$I(t) = \begin{cases} (0, ar(head(t))) & \text{if } a \text{ occurs in } t \\ (1, ar(head(t))) & \text{otherwise} \end{cases}$$

With this index function it is trivial to find a training set to learn a flat representative term space map to recognize the above set. It is obviously of less value for more general problems. Note, however, that such a construction can be used to integrate prior knowledge into a term space map.

As we have seen in the proof for Theorem 6.3, flat term space maps may need very large training sets to learn even simple concepts. The reason for this is that they do not

generalize beyond the level defined by the index function. *Recursive term space maps* are an attempt to alleviate this problem to some degree by considering different parts of the term individually.

Definition 6.12 (Representative recursive term space maps)
Consider M, t, *eval* and I as in definition 6.10. We extend *eval* to the multi-set of subterms of terms in M by associating $eval(t)$ with all occurrences of subterms of t. The *representative recursive term space map* for I, *eval* and M is the representative basic term space map

$$rrtsm_I(M) = \{(i, e(i, I, M), (tsm_{i,1} \dots tsm_{i,ar_I(i)}) | i \in I(M)\}$$

where

$$e(i, I, M) = \frac{\sum_{t \in \{t' \in M | I(t') = i\}} eval(t)}{|\{t' \in M | I(t') = i\}|}$$

as for representative basic term space maps, and where

$$tsm_{i,j} = rrtsm_I(\{(t|_j \mid t \in M, I(t) = i\})$$

for all $i \in I(M), j \in \{1, \dots, ar_I(i)\}$.

◀

A recursive term space map can be seen as a structure that reflects the way a term can be constructed by selecting one of multiple *alternatives* at the root position and then continuing this process for each of the subterms. It can be constructed by recursively partitioning a multi-set of evaluated terms.

To make use of the recursive nature of the term space map, we also need to evaluate terms recursively. Intuitively, the evaluation of a term under a recursive term space map is the average evaluation of all its subterms under the corresponding basic term space maps.

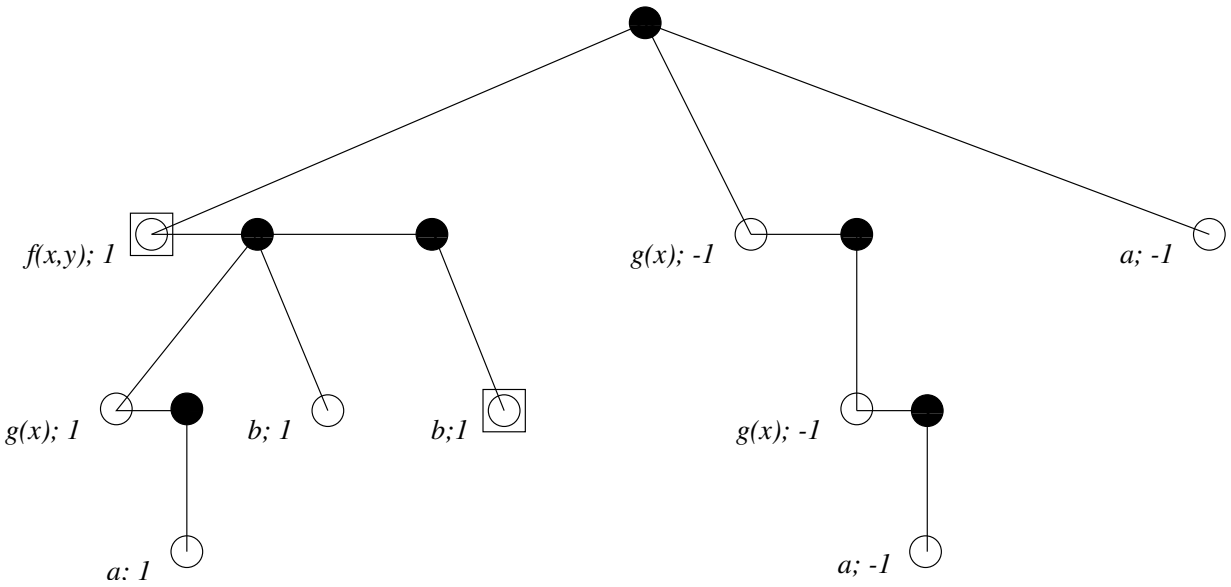
Definition 6.13 (Recursive term evaluation)
Let t be a term, let *tsm* be a basic term space map for some index function I and assume a constant value $e_u \in \mathbf{R}$ (the *evaluation of unmapped terms*).

- The *recursive evaluation weight of t under tsm* is defined as follows:

$$revw(tsm, t, e_u) = \begin{cases} e + \sum_{i=1}^{n} revw(tsm_i, t|_i, e_u) & \text{if there exists a TSA} \\ & (i, e, (tsm_1, \dots, tsm_n)) \\ & \text{with } i = I(t) \text{ in } tsm \\ e_u + \sum_{i=1}^{n} revw(\{\}, t|i, e_u) & otherwise \end{cases}$$

- The *recursive evaluation of t under tsm* is

$$rev(tsm, t, e_u) = \frac{revw(tsm, t, e_u)}{|O(t)|}$$



Remarks: See Fig 6.3. Boxed alternatives are used in the evaluation of $f(a, b)$, the subterm a has the default evaluation.

Figure 6.4: A representative recursive term space map

◀

Recursive term space maps for the index function I_{ar} subsume *term evaluation trees* as described in [Sch95, DS96a]. Let us again consider an example.

Example: Consider the set of terms from the example on page 92. Figure 6.4 shows a graphic representation of $rrtsm_{I_{top_1}}(M)$.
It is $rev(rrtsm_{I_{top_1}}(M), f(a, b), 0) = \frac{1+0+1}{3} = \frac{2}{3}$.

Both flat and recursive term space maps allow an evaluation or classification of infinite sets of terms. However, the evaluation is only based on a finite initial part of the term. Term nodes at positions deeper than any nodes in terms from the training set, for example, never contribute to an evaluation except possibly with the unmapped evaluation value. While such features as maximal term depth or absolute position of subterms can be easily represented, position-independent features (e.g. occurrence of a certain subterm or function symbol) can only be learned for finite classes of terms.

Recurrent term space maps overcome this restriction by using a single global partitioning for the evaluation of all terms and subterms.

Definition 6.14 (Representative recurrent term space maps)
Consider again M, t, *eval* and I as in definition 6.10.

- The *flattened term representation* of a term t is the multi-set $ftr(t) = \{t|_p \mid p \in O(t)\}$. If the term t has an associated evaluation $eval(t)$ we also associate this value with all terms in $ftr(t)$.
- The *flattened term set representation* of a set or multi-set of terms M is the multi-set $ftsr(M) = \cup_{t \in M}\ ftr(t)$.
- The *representative recurrent term space map* for I, *eval* and M is the basic term space map $rctsm_I(M) = rftsm_I(ftsr(M))$.

◀

As with recursive term space maps, we need a new way to evaluate terms. We again want to assign the average evaluation of all subterms to a term. However, we only use a single term space map now and *recurrently* apply it to evaluate all subterms.

Definition 6.15 (Recurrent term evaluation)
Let t be a term, let *tsm* be a basic term space map for some index function I and assume a constant value $e_u \in \mathbf{R}$ (the *evaluation of unmapped terms*).

- The *recurrent evaluation weight of t under tsm* is defined as follows:

$$cevw(tsm, t, e_u) = \begin{cases} e + \sum_{i=1}^{n} cevw(tsm, t|_i, e_u) & \text{if there exists a TSA} \\ & (i, e, (tsm_1, \ldots, tsm_n)) \\ & \text{with } i = I(t) \text{ in } tsm \\ e_u + \sum_{i=1}^{n} cevw(tsm, t|_i, e_u) & otherwise \end{cases}$$

- The *recurrent evaluation of t under tsm* is

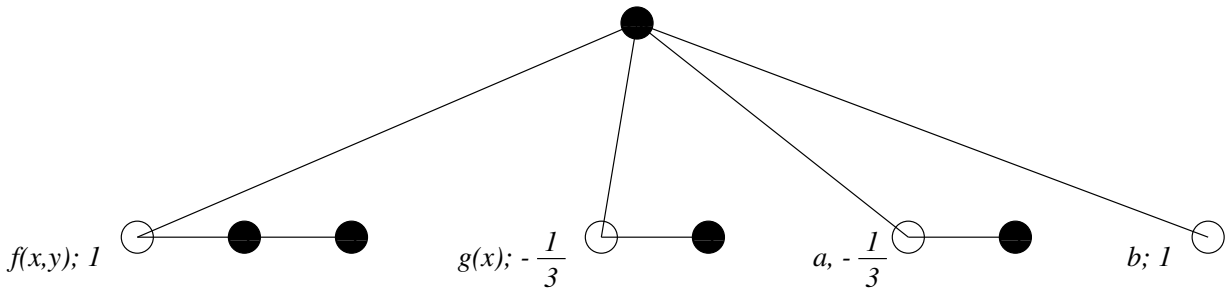
$$cev(tsm, t, e_u) = \frac{cevw(tsm, t, e_u)}{|O(t)|}$$

◀

Let us again return to our previous example:

Example: Consider the set of terms from the example on page 92. It is
$ftr(f(g(a), b)) = \{f(g(a), b), g(a), a, b\}$,
$ftr(f(b, b)) = \{f(b, b), b, b\}$,
$ftr(g(g(a))) = \{g(g(a)), g(a), a\}$ and
$ftr(a) = \{a\}$.
Figure 6.5 shows a graphic representation of $rctsm_{I_{top_1}}(M)$. As an example, the evaluation of the TSA indexed by a is computed as $\frac{1+-1+-1}{3} = -\frac{1}{3}$ for the three occurrences of a in $ftr(f(g(a), b))$, $ftr(g(g(a)))$ and $ftr(a)$, respectively.

It is $cev(rctsm_{I_{top_1}}(M), f(a, b), 0) = \frac{1+-\frac{1}{3}+1}{3} = \frac{5}{9}$.



Remarks: See Fig 6.3.

Figure 6.5: A representative recurrent term space map

If we employ a recurrent term space map with the index function I_{top_1}, the term space map basically maps each subterm to its top function symbol, and the evaluation of a term t is the average evaluation of all function symbol occurrences in t. If we use the TSM-evaluation to modify a basic term weight, this is very similar to the effect of learning different weights for different function symbols as described in [Fuc95b].

6.4 Dynamic Selection of Index Functions

We have, up to now, only considered term space maps based on a predefined index function. However, a suitable index function may be hard to recognize *a priori* if we are confronted with a unknown learning problem. We will now use some concepts from *information theory* ([SW49], see [RN95],pages 540–543 for a modern introduction or [SG95, Sch96] for a more rigorous modern description) and apply them to the selection of of an index function in a similar way as Quinlan used them for the selection of features tests in decision trees [Qui92, Qui96].

The amount of information we can get from an event is related to the probability of this event happening: If a predetermined event (with probability 1) happens, we have not gained any new information. If, on the other hand, an event with probability 0.5 happens, we have more information than before (in the particular case of $p = 0.5$, we have gained exactly one *bit* of information). If an experiment can yield a number of (disjoint) results, the expected amount of information we will gain from it is described by the *entropy* of the probability distribution over the possible results. The following definitions formalize these concepts:

Definition 6.16 (Information, Entropy)
Let $A = \{a_1, \ldots, a_n\}$ be an experiment with the possible results or observations $a_1, \ldots, a_n$ and let $P : A \to \mathbf{R}$ be a probability distribution on the results of A (i.e. $P(a_i)$ is the probability of observing a_i as the result of A).

- The amount of *information* gained from the observation of an event (or the *information content* of this event) is $J(a_i) = -log_2(P(a_i))$.
- The *entropy* of A is the expected information gain of performing A,

$$H(A) = \sum_{i=1}^{n} P(a_i)J(a_i)$$

◀

Different experiments may not be fully independent from each other. Consider e.g. the two experiments "determine if a person is a soccer fan" and "determine the gender of a person". While the result of the first experiment does not fully determine the outcome of the second one, it does lead to a different expectation about it. In general, whenever we perform more than one experiment, we do potentially lower the information content we can gain from all but the first experiment, i.e. we lower the remaining entropy of these experiments. This is the principle behind testing certain features to determine the class of an object: If the feature distribution is in some way related to the class distribution, getting information about a feature also gets us some information useful for determining the class.

Definition 6.17 (Conditional information, Conditional entropy)
Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ be two experiments with probability distributions P_a and P_b.

- $P(a|b)$ is the *conditional probability* of a under the condition b.
- The *conditional information content* of a result a_i under the condition b_j is $J(a_i|b_j) = -log_2(P(a_i|b_j))$
- The *conditional entropy* of A under the condition b_j is

$$H(A|b_j) = \sum_{i=1}^{n} P(a_i|b_j)J(a_i|bj)$$

◀

If we want to determine the value of B for getting information about the outcome of A, we again need to average over the possible outcomes of B to determine the *remaining entropy* of A. Our expected *information gain* from the experiment B then is exactly the difference in the entropy of A and the remaining entropy of A after performing B:

Definition 6.18 (Remainder entropy, Information gain)
Let once more $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ be two experiments with probability distributions P_a and P_b, respectively.

- The *remainder entropy* of A (after performing B) is

$$H(A|B) = \sum_{j=1}^{m} P_b(b_j) H(A|b_j)$$

- The *information gain* of performing B (to determine the result to A) is $G(A|B) = H(A) - H(A|B)$.

◀

In our case, the experiment A is the classification of a term. The experiment B is the result of applying an index function to the term. As we do not know the real probabilities of the outcomes of either of the experiment, we use the relative frequencies of the outcomes in the training set as estimates for the probabilities. This is similar to the work done by Quinlan on *top down induction of decision trees*. Quinlan uses feature tests on finite feature vectors as experiments and then proceeds to select the feature that yields the highest information gain as the first feature to test. In Quinlan's case, the different experiments are at the same level of abstraction, and are at least conceptually independent.

In our case, where we want to select one among multiple index functions, this approach has to be adapted. Our index functions are not even conceptually independent, and represent very different levels of abstraction. If we consider a finite set M of preclassified terms (without noise), it is obvious that e.g. the index function I_{top_k} (compare page 88) for a value of k that is larger than the depth of the deepest term in M will split the set into individual terms and will thus immediately yield all the information for a correct classification. This partition of M will, however, allow no generalization to term outside of M. Our aim is to find a balance between information gain and generality of the index function. We therefore introduce the *relative information gain*. The relative information gain sets the information gain towards the desired classification in relation to the expected amount of additional information necessary to perform the test.

Definition 6.19 (Relative information gain)
We again assume $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ with probability distributions P_a and P_b, respectively. The *relative information gain* of performing B (to determine the result to A) is

$$R(A|B) = \frac{H(A) - H(A|B)}{H(B) - (H(A) - H(A|B))}$$

◀

Term space maps are used primarily for evaluation, not for classification, and we therefore cannot expect the training sets to come preclassified. However, our aim for both classification and evaluation is to differentiate between terms based on their evaluation. Hence, we create two artificial classes of terms, those with a high evaluation and those with a low evaluation. In the case of preclassified terms (were terms from one class have e.g.

evaluation -1 and terms from the other class evaluation 1), the artificial classes we create coincide with the given classes.

Definition 6.20 (Relative information gain for index functions)
Let M be a multi-set of terms t with evaluations $eval(t)$ and let

$$l = \frac{\sum_{t \in M} eval(t)}{|M|}$$

be the average evaluation of terms in M. We partition the term set into two classes, $M = M^+ \uplus M^-$ with $M^+ = \{t \in M | eval(t) > l\}$ and $M^- = \{t \in M | eval(t) \leq l\}$. Let A be the experiment of selecting a term from M with the two outcomes $t \in M^+$ and $t \in M^-$ with the associated relative frequencies

$$p^+ = \frac{|M^+|}{|M|} \text{ and } p^- = \frac{|M^+|}{|M|}$$

Now let I be an index function. It defines an experiment B with the possible outcomes in $I(M)$, where each result i has the associated probability

$$p_i = \frac{|\{t \in M | I(t) = i\}|}{|M|}$$

Then the *relative information gain induced by I on M* is $R(I, M) = R(A|B)$

◀

We can now use these definitions to build term space maps that select the index function that gives them the best relative information gain.

Definition 6.21 (Information-optimal index functions)
Let M be a multi-set of terms t with evaluations $eval(t)$ and let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of index functions.

- An *information-optimal index function from $\mathcal{I}$ for M* is an index function I from $\mathcal{I}$ with $R(I, M) = max\{R(I, M) | I \in \mathcal{I}\}$.
- An *information-optimal basic term space map (for a set $\mathcal{I}$ of index functions and a training set M)* is a basic term space map with respect to I and M where I is an optimal index function from $\mathcal{I}$.

◀

Flat and recurrent term space maps use only a single index function. However, for recursive term space maps, we can select an information-optimal index function for each sub-TSM:

Definition 6.22 (Information-optimal term space maps)
Consider M, t, *eval* (extended to subterms) and I as in Definition 6.12. Let further $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of index functions.

- An *information-optimal flat term space map* for M and $\mathcal{I}$ is any term space map $tsm \in \{rftsm_I(M) \mid I \text{ is information-optimal for } M \text{ in } \mathcal{I}\}$.
- An *information-optimal recursive term space map* for M and $\mathcal{I}$ is an information-optimal basic term space map tsm for $\mathcal{I}$ and M,

$$tsm = \{(i, e(i, I, M)(tsm_{i,1} \ldots tsm_{i,ar_I(i)}) | i \in I(M)\}$$

  where

$$e(i, I, M) = \frac{\sum_{t \in \{t' \in M | I(t') = i\}} eval(t)}{|\{t' \in M | I(t') = i\}|}$$

  and where the $tsm_{i,j}$ are information-optimal recursive term space maps for $\{(t|_j \mid t \in M, I(t) = i\}$ and $\mathcal{I}$ for all $i \in I(M), j \in ar_I(i)$.
- An *information-optimal recurrent term space map* for M and $\mathcal{I}$ is any term space map $tsm \in \{cftsm_I(M) \mid I \text{ is information-optimal for } ftsr(M) \text{ in } \mathcal{I}\}$.

◀

Note that an information-optimal recursive term space map is not necessarily an representative recursive term space map.

The experimental results in Chapter 8 show that the information-gain based selection of the index function is indeed able to find very good index functions at least for flat and recurrent term space maps.

6.5 Summary

In this chapter we have discussed the term-based learning algorithms that are central to a theorem prover that tries to learn clause evaluations. After an overview of existing approaches we have introduced term space mapping as a new class of fast, term-based learning algorithms that are able to learn numerical evaluations for terms. Term space mapping works by partitioning the space of all terms into partitions defined by an index function. We described three different schemes to perform this partition: Flat term space maps (subsuming memorization), recursive term space maps (subsuming term evaluation trees) and recurrent term space maps. Finally, we introduced the concept of *relative information gain*, based on the entropy of class distributions, to select the abstraction (represented by an index function) with the best relationship between information gain and generality.

Chapter 7

The E/TSM ATP System

We will now describe the learning theorem prover E/TSM that implements the concepts introduced in the previous chapters. It stores proof experiences represented as sets of annotated clause patterns in a knowledge base and uses these experiences to create a heuristic clause evaluation function that is used to select the *given clause* in new proof searches.

Figure 7.1 shows the overall architecture of the proof system. The prover is build around five major components:

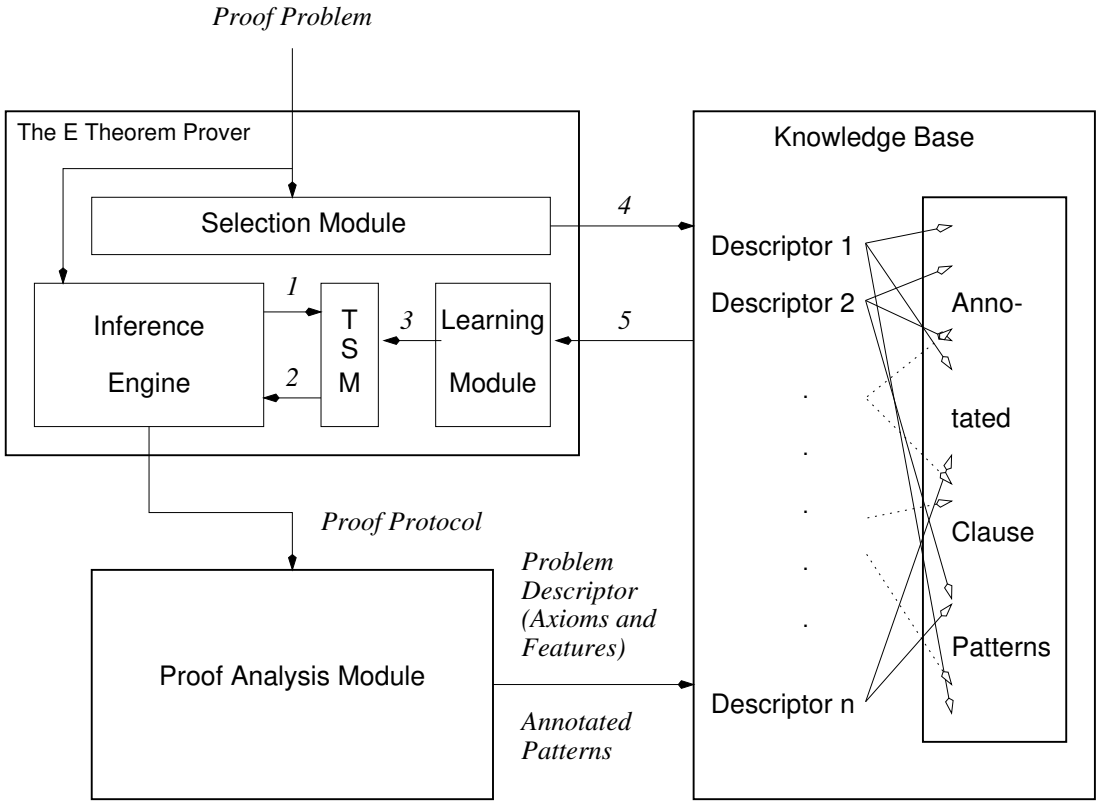
- The *inference engine* realizes the **SP** calculus described in Section 2.7. It is capable of writing an abbreviated protocol of these inferences. For details about the conventional features of the inference engine see Appendix A.

- The abbreviated protocol is interpreted by an independent *proof analysis* program. It is expanded and directly translated into a proof derivation graph as described in Section 5.3. Based on this graph, the proof search is transformed into a set of annotated clauses.

- The central repository of learned knowledge is the *knowledge base*. It contains preprocessed and indexed annotated patterns organized for efficient selection of knowledge from suitable proof examples. We describe the organization of the knowledge base in Section 7.1 in this chapter.

- The selection of knowledge from the knowledge base is controlled by a *selection module*. It analyzes new proof problems and requests search control knowledge about similar proof problems from the knowledge base. Section 7.2 describes the details.

- Finally, this knowledge is compiled by the *learning module* into a *term space map* and used to evaluate new clauses. We describe this process in sections 7.3 and 7.4.



1 *New clauses*

2 *Evaluations*

3 *Compiled term space map*

4 *Request (feature vector, arity frequency vectors)*

5 *Selected clause patterns (with annotations)*

Figure 7.1: Architecture of E/TSM

7.1 The Knowledge Base

The knowledge base is at the core of the learning system. Data generated from proof experiences is stored in a compact form and indexed for efficient access and processing. A knowledge base consists of 4 distinct parts: The *knowledge base description*, the *problem index*, the *clause pattern store* and the *proof experience archive*.

The *knowledge base description* determines how proof experiences are analyzed and preprocessed. There are two parts of the knowledge base description: A partial signature *sig* that describes a set of function symbols F_f and a set of parameters for the proof analysis process. Relevant parameters in the current implementation are the proportion of positive

and negative clause families selected from each proof experience, the number of examples to extract from failed proof searches, and a flag that determines if all clauses of a clause family are used to represent it or whether only the evaluated clauses are chosen. The set F_f contains function symbols with known and fixed intended semantic from a domain of interest. These symbols are not generalized in the clause pattern representation of the proof experiences.

The *problem indexproblem index* associates each individual proof experience with a unique identifier *proof_id* and an experience descriptor, consisting of a description of the signature and a vector of numerical features. For details see the next section.

The *clause pattern store* stores the representative clauses selected from the proof experiences. It contains a *recursive representative pattern* (with respect to the symbols in F_f described in the signature) for each clause selected from a proof experience. These clauses are represented as a set of maximally shared representative patterns (compare Definition 6.3 and Section A.1.1), and each clause pattern is associated with a set of annotations describing the properties of the corresponding clauses in different proof searches. A single annotation in the clause store is of the form *proof_id:(pd,mp,mn,gp,gn,sc)*, where *proof_id* is an identifier for the original proof experience as described above and the other values represent the effect of the clause in the proof search as described in Section 5.3.2.

The *proof experience archive*, finally, stores the unprocessed proof experiences, represented by the original problem specification and the set of annotated, but not generalized, clauses selected to represent the corresponding proof search. This archive is only used for manual verification and analysis, or for the reuse of the proof experiences in new knowledge bases. It is not actively used by the learning component.

Knowledge bases are managed by a set of programs for knowledge base creation, insertion of new proof experiences, and deletion of existing proof experiences.

7.2 Proof Example Selection

We will now discuss the experience selection component of the theorem prover. Its task is to select a subset of proof experiences that are *similar* to a new problem from the knowledge base. To achieve this, we represent each proof problem by three vectors of numbers and use distance functions to define a concept of similarity. A proof problem is represented by a distribution of the number of function symbols of different arities, a similar distribution for predicate symbols, and a vector of numerical features representing properties of the problem specification.

Let us first consider the numerical signature representations and the distance measure it induces on signatures:

Definition 7.1 (Arity frequency vector, Signature distance)

- Let $sig = (F, ar)$ be a signature and assume $n \in \mathbf{N}$. The *arity frequency vector* for sig and n is the vector $af(sig, n) = (|\{f \in F \mid ar(f) = 0\}|, \ldots, |\{f \in F \mid ar(f) = n\}|)$.
- Let $sig_1 = (F_1, ar_1)$ and $sig_2 = (F_2, ar_2)$ be two signatures and assume $n \in \mathbf{N}$,

$n = max(ar_1(F_1) \cup ar_2(F_2))$. The *signature distance* between sig_1 and sig_2 is $sd(sig_1, sig_2) = \overline{rdist_E}(af(sig_1, n), af(sig_2, n))$.

◀

In addition to the signature composition, we also use clause features to describe a problem specification. Table 7.1 shows the 15 features used by E/TSM. Most of the features used are variations or generalizations of features used successfully for similar tasks in existing theorem provers. However, we have introduced the *standard deviation* of term or clause features as a new useful feature for a clause set (see [SB99]).

Feature	Description
f_1	Number of unit clauses in the clause set
f_2	Number of non-unit Horn clauses
f_3	Number of non-Horn clauses
f_4	Average term depth of terms in positive literals
f_5	Standard deviation of the term depth of terms in positive literals
f_6	Average term depth of terms in negative literals
f_7	Standard deviation corresponding to f_6 (see f_4, f_5)
f_8	Average term weight of terms (with function symbol weight $w_f = 2$, variable weight $w_v = 1$) in positive literals
f_9	Standard deviation corresponding to f_8
f_{10}	Average term weight of terms in negative literals
f_{11}	Standard deviation corresponding to f_{10}
f_{12}	Average number of positive literals in a clause
f_{13}	Standard deviation corresponding to f_{12}
f_{14}	Average number of negative literals in a clause
f_{15}	Standard deviation corresponding to f_{14}

Table 7.1: Clause set features used for example selection

Definition 7.2 (Feature representation of formulae)
Let $\mathcal{F}$ be a formula in clause normal form. We call the vector $frep(F) = (f_1, \ldots, f_{15})$, where the f_i are computed as described in Table 7.1, the *feature representation* of $\mathcal{F}$.

◀

Given these definitions, we will now define a *distance measure* on proof problem specifications. While in principle a *weighted distance measure* is a more general approach to measuring the distance between two vectors, we have opted for the simple relative Euclidean distance instead. Adding weights for all features leads to an extremely large parameter space that cannot be explored with reasonable resources. Moreover, the preliminary experiments performed with DISCOUNT/TSM [Bra98, SB99] showed that at least for the unit-equational case and the feature selection used by DISCOUNT, the best results were

obtained with equal weighting of all features. We therefore currently restrict ourselves to this simpler case.

Definition 7.3 (Proof problem distance)
Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two formulae in clause normal form, let sig_{P1} and sig_{P2} be two signatures describing exactly the predicate symbols occurring in $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively, and let similarly sig_{F1} and sig_{F2} be two signatures describing exactly the function symbols occurring in $\mathcal{F}_1$ and $\mathcal{F}_2$. The *distance* between the two formulae is $specd(\mathcal{F}_1, \mathcal{F}_2) = sd(sig_{P1}, sig_{P2}) + sd(sig_{F1}, sig_{F2}) + \overline{rdist_E}(frep(F_1), frep(F_2))$.

◀

The selection module of E/TSM computes the proof problem distance between a new problem $\mathcal{F}$ and all problems corresponding to stored proof experiences. It then selects the subset of problems with the smallest distance to $\mathcal{F}$. The size of this set can be limited by either giving a total number of problems to select, by giving a percentage of problems to select, or by stating a limit on the distance (given in units of the *average distance*) for problems to be selected. As stated above, proof problems are represented by pre-computed arity frequency vectors and feature representations in the problem index. This speeds up the selection process significantly and minimizes expensive I/O operations.

7.3 The Learning Module

The task of the learning module is to compile the selected knowledge into a term space map for clause evaluation. To perform this task, we have to transform the annotated clause patterns from the clause store into terms with an associated evaluation, and feed them into the term space mapping algorithm. We therefore have to find an evaluation for each clause pattern based on the annotation vectors it carries.

Input to the learning module is the complete clause store and a list of proof problem identifiers corresponding to problems determined by the selection module. The learning module then selects all clause patterns with at least one annotation from the set of selected proof experiences and strips off all annotations from other experiences.

In the resulting set, each clause annotation corresponds to an occurrence of a clause in a selected proof experience. However, we do of course want to avoid to actually create a copy of the clause pattern for each annotation. Therefore, we first combine the individual annotations, and then transform the result into a tuple $(no, \mathcal{P}, eval)$, where no is the number of clause occurrences represented by the clause pattern $\mathcal{P}$ and $eval$ is the resulting evaluation.

Recall Section 5.3.2, where we discussed potentially important properties of the role a clause played in the proof search. We have encoded the clause properties that can be determined from a single proof search in the individual annotations. However, we have not yet encoded the number of proofs to which clauses corresponding to a pattern have contributed, as this feature can only be determined after we have selected a relevant set

of proof experiences. We now combine all annotations for a clause patterns in a single annotation.

Definition 7.4 (Combined clause pattern annotation)
Let $a_1, \ldots, a_n$ be a set of annotations for a clause pattern, where each a_i is of the form $proof\_id_i : (pd_i, mp_i, mn_i, gp_i, gn_i, sc_i)$. The *combined clause pattern annotation* for this clause is the vector $(s, pn, \overline{pd}, \overline{mp}, \overline{mn}, \overline{gp}, \overline{gn}, \overline{sc})$, where $s = n$ is the number of original annotations, $pn = |\{a_i \mid pd_i = 0, i \in \{1, \ldots, n\}\}|$ is the number of proofs in which the pattern participated, and the other values are the average values of the corresponding entries in the original annotations.

◀

The individual features in a combined annotation typically have widely varying ranges. However, we want to be able to control both the influence of each of these parameters in the final evaluation and the range of the final evaluation independently. We therefore first *normalize* each value in the annotation vector to a value between 0 and 1. We then use a linear combination of the values in a normalized annotation as the final evaluation for a clause pattern, and again normalize this final evaluation over the set of evaluated clause patterns.

Definition 7.5 (Experience-based clause pattern evaluation)
Let $M = \{(\mathcal{P}_1, a_1), \ldots, (\mathcal{P}_n, a_n)\}$ be a set of clause patterns with combined annotations, where each combined annotation has the form
$a_i = (s_i, pn_i, pd_i, mp_i, mn_i, gp_i, gn_i, sc_i)$.

- Assume
  - $\hat{pn} = max\{pn_i | i \in \{1, \ldots, n\}\}$
  - $\hat{pd} = max\{pd_i | i \in \{1, \ldots, n\}\}$
  - $\hat{mp} = max\{mp_i | i \in \{1, \ldots, n\}\}$
  - $\hat{mn} = max\{mn_i | i \in \{1, \ldots, n\}\}$
  - $\hat{gp} = max\{gp_i | i \in \{1, \ldots, n\}\}$
  - $\hat{gn} = max\{gn_i | i \in \{1, \ldots, n\}\}$
  - $\hat{sc} = max\{sc_i | i \in \{1, \ldots, n\}\}$

  The *normalized annotation* corresponding to a combined annotation $a = (s, pn, pd, mp, mn, gp, gn, sc)$ in M is the vector

$$norm_M(a) = (n, \frac{pn}{\hat{pn}}, \frac{pd}{\hat{pd}}, \frac{mp}{\hat{mp}}, \frac{mn}{\hat{mn}}, \frac{gp}{\hat{gp}}, \frac{gn}{\hat{gn}}, \frac{sc}{\hat{sc}})$$

- Now assume a vector of weights $w = (w_{pn}, w_{pd}, w_{mp}, w_{mn}, w_{gp}, w_{gn}, w_{sc})$.

– The *evaluation* of $(\mathcal{P}, a) \in M$ is

$$eval_M((\mathcal{P}, a), w) = w_{pn}\frac{pn}{\hat{pn}} + w_{pd}\frac{pd}{\hat{pd}} +$$

$$w_{mp}\frac{mp}{\hat{mp}} + w_{mn}\frac{mn}{\hat{mn}} + w_{gp}\frac{gp}{\hat{gp}} + w_{gn}\frac{gn}{\hat{gn}} + w_{sc}\frac{sc}{\hat{sc}}$$

– The *normalized evaluation* of $(\mathcal{P}, a) \in M$ is

$$neval_M((\mathcal{P}, a), w) = \frac{eval_M((\mathcal{P}, a), w) - min\ eval_M(M, w)}{max\ eval_M(M, w) - min\ eval_M(M, w)}$$

– The *evaluated clause pattern* corresponding to $(\mathcal{P}, a) \in M$ (where a is of the form $(s, pn, pd, mp, mn, gp, gn, sc)$) is the 3-tuple

$$ecp_M((\mathcal{P}, a)) = (s, \mathcal{P}, neval_M((\mathcal{P}, a), w)$$

◀

Note that as a consequence of this definition, the normalized clause evaluations have the following property:

Corollary: The normalized evaluation of a clause pattern is always an element of the set $[0; 1]$.

We have implemented the term space mapping algorithms in a way that they internally treat an evaluated clause pattern $(s, \mathcal{P}, e)$ as a multi-set containing s instances of $\mathcal{P}$ with evaluation e. The overall task of the learning module hence is realized by transforming the annotated clause patterns into evaluated clause patterns and feeding the resulting set into a term space mapping algorithm.

Our implementation of term space mapping supports flat, recursive and recurrent term space maps. It also can, on request of the user, recode the recursive clause patterns stored in the knowledge base *on the fly* into flat clause patterns. We have implemented all index functions described in Theorem 6.1 (up to a compile-time defined depth limit for the index functions based on top terms). Index functions can be selected by the user. Alternatively, the cheapest (in term of computing resources for the evaluation of new clauses)[1] information optimal index function is automatically selected.

There is one more open variable we need to determine before we can use the term space map to evaluate new clauses. This is the value e_u assigned to term nodes not mapped by the term space map. For arbitrary classification experiments it makes sense to use the

[1]These leads to the following preference ordering on index functions: I_{ar}, I_{symb}, I_{id}, I_{top_1}, $I_{top'_1}$, I_{cstop_1}, I_{estop_1},..., I_{top_n}, $I_{top'_n}$, I_{cstop_n}, I_{estop_n}. In general, the explicit copying of term tops is the most expensive operation in index function computation in our current implementation.

average evaluation of all terms in the training set used to build the term space map, on the assumption that this training set is a typical sample. However, for clause evaluation we can use more a-priori knowledge. Assume a hypothetical clause that maps onto a node not mapped by the training set. Obviously, this clause does not participate in any proof in the set of selected proof experiences. Hence, its value for the parameter *pn* is 0. Similarly, this fictive clause is not even close enough to a proof to be selected. As an estimate, we can say that the value *pd* for this clause is greater than for any clause that has been selected as part of a proof representation. For the other parameters, the best estimates we can give are the average values of the clauses in the training set. This leads to following premise:

> **Premise:** A good (normalized) evaluation e_u for unmapped term nodes in a TSM generated by a set of clauses with combined evaluations M is the (normalized) evaluation of a fictive clause with combined evaluation (1, 0, *pd*, *mp*, *mn*, *gp*, *gn*, *sc*), where *pd* is the maximum value of any proof distance in M plus 1, and where *mp*, *mn*, *gp*, *gn* and *sc* are the weighted averages of the corresponding values in M.

7.4 Knowledge Application

In the application phase, the term space map generated by the learning module is used to define a heuristic evaluation function for new clauses. The complete application phase consists of transforming newly generated and normalized clauses into representative clause patterns and evaluating them with the help of the term space map.

The resulting heuristic evaluation function should have two important properties. First, the resulting search strategy should be *complete*, i.e. any proof derivation generated with this strategy should be fair. Secondly, the search strategy should perform adequately even in cases where no previous knowledge exists.

The term evaluation functions induced by a term space map constructed as described in the previous section evaluates a clause pattern based *only* on information from previous proof searches, not on syntactical properties or on the current proof derivation. If the term space map covers only a finite subset of all clauses (as e.g. is the case for any representative flat term space map for the index function I_{id}), the search strategy on new clauses is undefined (or rather defined by the implementation of the base prover and usually similar to FIFO), and unlikely to deliver good performance.

Moreover, if a more general index function is used, the evaluation function defined by a term space map may even result in *unfair* proof derivations:

Example: Consider the flat term space map (for index function I_{ar} and flat representative clause patterns) in Figure 7.2, which might be the result of a proof search where only unit clauses contributed to the proof. It will always yield a lower evaluation for unit clauses than for clauses with 2 literals, and hence will not generate a fair search derivation for most problems.

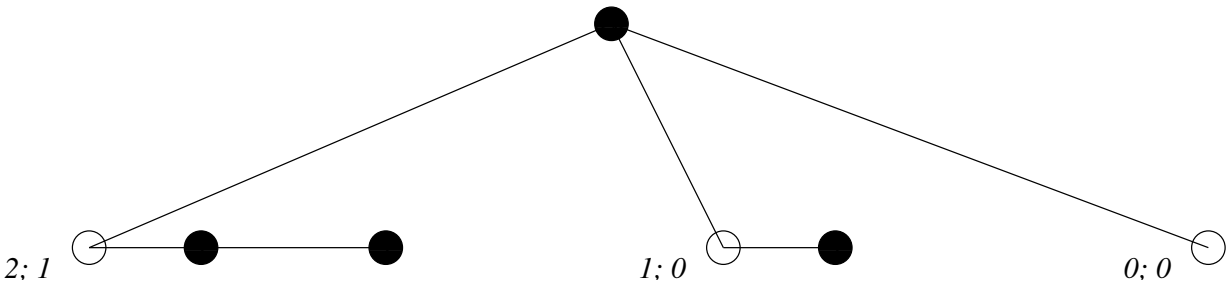


Figure 7.2: A flat TSM representing an unfair term evaluation function

To overcome this problem, we combine the TSM defined evaluation with a conventional clause weight evaluation:

Definition 7.6 (The *TSMWeight* evaluation function)
Let *tsm* be a term space map, let $tev \in \{fev, rev, cev\}$ be a TSM-based evaluation function, assume $e_u \in [0;1]$, $w_{learn} \in \mathbf{R}^+$ (the *weight of the learned evaluation*) and $w_f, w_v \in \mathbf{R}$. Let $\mathcal{C}$ be clause and let $\mathcal{P}$ be a (flat or recursive) representative clause pattern for $\mathcal{C}$. Then

$$TSMWeight(tsm, tev, e_u, w_{learn}, w_f, w_v, \mathcal{C})$$

$$= (1 + w_{learn}\ tev(tsm, \mathcal{P}, e_u)) \times CWeight(w_f, w_v, \mathcal{C})$$

◀

E/TSM uses this evaluation function for guiding the proof search. Note hat for the value $w_{learn} = 0$ the evaluation function is equivalent to the ordinary clause weight, and that for large values of w_{learn} it becomes equivalent to the product of the term space map evaluation and the clause weight evaluation.

This evaluation function results (under reasonable assumptions about the parameters) in a fair proof derivation:

Theorem 7.1 (Fairness of *TSMWeight*)
Let *tsm* be a representative TSM for a set of clauses with normalized evaluations and assume *tev*, e_u, w_{learn}, w_f and w_v as in the previous definition with the additional constraints that $w_f > 0$ and $w_v > 0$. Then any proof derivation resulting from the *given-clause* algorithm that always selects the clause $\mathcal{C}$ with the lowest weight *TSMWeight*(*tsm*, *tev*, e_u, w_{learn}, w_f, w_v, $\mathcal{C}$) is fair.

Proof: Note that the normalized clause evaluations as well as e_u are from the interval $[0;1]$. Hence all evaluations of terms under the representative TSM are also from this interval.

According to Theorem 4.1, we have to show that no persistent clause remains in the set U forever. Now assume that a clause $\mathcal{C}$ with evaluation $e = TSMWeight(tsm, tev, e_u, w_{learn}, w_f, w_v, \mathcal{C})$ is never selected for processing. Obviously, only clauses with a clause weight smaller than e can get a smaller overall evaluation than $\mathcal{C}$. But there is only a finite number of clauses with this property, as the number of symbol occurrences in the clause is limited by $\frac{e}{min(w_f,w_v)}$ and the number of different function symbols and variables[2] is finite as well.

■

7.5 Summary

In this chapter we have described how we integrated the techniques developed in the previous chapters to create the learning theorem prover E/TSM. This prover is the first ATP system for clausal logic that combines solutions to all major problems for learning from previous proof experiences.

The proof system consists of five major components: The inference engine, the proof analysis module, the knowledge base, the proof experience selection module, and the learning module.

Proof search experiences are generated by the inference engine. The proof analysis module transforms each proof search into a compact representation, consisting of a numerical representation of the proof problem and a set of annotated clause patterns corresponding to important search decisions. Annotations for a clause pattern store information about the usefulness of the corresponding clause in a given proof search, encoded as a numerical vector. The representations of each proof problem are then stored in a knowledge base.

To apply the stored knowledge to a new proof search, the proof experience selection module analyzes the new formula, transforms it into a numerical representation, and then uses the normalized Euclidean distance between this representation and the representations of previous proof problems in the knowledge base to select a subset of similar proof experiences.

The learning module accepts the list of similar proof problems as input. The clause patterns corresponding to the selected problems are associated with an evaluation that is computed from their annotations, and are transformed into a representative term space map. This term space map is then used to modify a conventional evaluation heuristic that is used to guide the proof search.

The resulting proof derivation is always fair, and hence the completeness of the prover is not compromised by the use of learning heuristics. The following chapter shows that very good results can be achieved with the learning heuristics.

[2]As stated in Section 2.5, we identify clauses that are only variants of each other, and hence can assume a clause to be in a variable-normalized form.

Chapter 8

Experimental Results

In this chapter we present the experimental results obtained with our implementation. In the first part, we apply term space mapping to some artificial classification experiments and demonstrate that the different kind of term space maps can learn different properties of terms. In the second part, Section 8.2, we present the results obtained by our learning theorem prover E/TSM under different conditions. We use a set of relatively easy proof problems as training examples and show that the learned search control knowledge helps us to find proofs for new, harder problems.

8.1 Artificial Classification Problems

To be able to evaluate the performance of the different versions of term space mapping, we have created a set of simple test problems. For each of this problems, a set of terms is split into two classes, a positive class and a negative class. A term space map is trained on a part of the total set, and then used to classify the remaining terms. We used all three kinds of term space maps, and all index functions identified in Theorem 6.1, with a depth limit of 5 for the index functions based on term tops.

8.1.1 Experimental Setup

Random term generation

The first problem in designing these classification experiments is to obtain suitable term sets. As the set of all terms is infinite for any non-trivial signature, there is no obviously fair probability distribution. Even naive term generation schemes will not terminate stochastically, as the number of potential open branches in a term increases exponentially with the term depth.

We have developed a procedure that handles these problems by a suitable distribution on a the arity of function symbols and by introducing a depth-based probability for term branch cut-off. The procedure is depicted in Figure 8.1.

Variables:

`l`	*Magical* number, yields distribution of arities. Larger values lead to broader terms. We use `l` = 0.95 in our experiments.
`d`	The depth level for which the term is generated, initially 0.
`t`	The newly generated term.
`s`	The top symbol of the generated term.
`a`	Number of direct subterms of the term to be generated.
$t_1 \ldots t_a$	Subterms.
`sig`	The signature (see main text).

```
procedure genterm(d)
{
  select a randomly from [0,3] with probabilities
      p(3) =          (1-1/(d+1))*(l*l/2*l/4)
      p(2) =          (1-1/(d+1))*(l*l/2*(1-l/4))
      p(1) =          (1-1/(d+1))*(l*(1-l/2))
      p(0) = 1/(d+1)+ (1-1/(d+1))*(1-l);=-
    select s with equal probabilities
        from {s ∈ sig |ar(s) = a};
    (t1,...,ta) := (genterm(d+1),...,genterm(d+1));
    t := s(t1,...,ta);
    return t;
  }
```

Figure 8.1: Generating random terms

We have generated two sets of 20,000 pseudo-random ground terms[1] over the signature $\{f_{01}/0, \ldots, f_{04}/0, f_{11}/1, \ldots, f_{13}/1, f_{21}/2, \ldots, f_{22}/2, f_{31}/3\}$ by repeatedly using this procedure. The first set , term set A, was filtered for repeated terms, the second set, B, contains terms as generated by the procedure. Table 8.1 shows some statistical data on the term sets. We will use these sets as the basis for all classification experiments. For two tests (recognizing symmetry and *memorization*) we will modify the term sets (to ensure that the required properties for these experiments are present), for all other experiments we use the sets as they were generated.

Our method for generating terms has advantages and disadvantages. Advantages are

[1]Note that term space mapping does not distinguish between constants and variable symbols in terms on a syntactic basis – all symbols are treated equally.

Term set	A	B
Number of terms	20,000	20,000
Number of distinct terms	20,000	15,097
Average term depth	5.814	5.032
Maximal term depth	14	14
Average number of symbols	13.410	10.751
Maximal number of symbols	75	69

Table 8.1: Term sets used in classification experiments

that the generated terms cover a relatively range for both size and depth, and that function symbols for a given arity are distributed equally. Disadvantages are that the method favors small terms heavily, and that function symbols of different arities are selected with very different probabilities. These properties are likely to skew some experiments.

Evaluation method

To get statistically significant results, we use *10 fold cross-evaluation*. This is a standard evaluation technique used in the field of machine learning. The complete set of all examples is randomly split into 10 equal parts (or *folds*). One after the other, each of the ten folds is set aside as a *test set*. The remaining 9 parts are combined into a *training set*. Terms in the training set are associated with the evaluation +1 for terms from the positive class and -1 for terms from the negative class. The terms are then used to compute a representative term space map. This TSM is then used to classify the part set aside as a test set. For each of the 10 experiments the rate of success is measured. The overall performance of the term space map is given by the average rate of success, reliability of this result is the standard deviation of the individual rates.

We compare the results with those that would be achieved of a *random guesser* and by a *naive learner*. A random guesser guesses the two classes randomly according to their frequency in the data set. If the relative frequency of the larger class is given by p, it achieves the average correctness of $p \times p + (1-p) \times (1-p)$. A naive learner always guesses the most frequent class. If the relative frequency of the larger class is again given by p, the expected correctness of a naive learner is p.

Classification limits

To classify terms with a term space map, we need to decide on an evaluation for unmapped terms (see Definitions 6.9, 6.13 and 6.15) and on the classification limit used to separate the two classes (Definition 6.11).

We use the average evaluation of all terms in the *training set* under the term space map as an evaluation for unmapped terms. As a term space map that is representative for a certain set of terms always maps all terms of this set completely, this value is well-defined.

For the classification limit, we choose the average of the average evaluations for terms in each of the two classes.

More formally, consider a multi-set $M = M^+ \uplus M^-$ of terms (*training examples*) and let *tsm* be a flat, recursive or recurrent representative term space map for M (with evaluations) and some index function, and let *tev* be the corresponding (flat, recursive or recurrent) evaluation function.

We use the value e_u for the evaluation of unmapped terms and the classification limit l given by

$$e_u = \frac{\sum_{t \in M} tev(tsm, t, 0)}{|M|}$$

and

$$l = \frac{\frac{\sum_{t \in M^+} tev(tsm,t,e_u)}{|M^+|} + \frac{\sum_{t \in M^-} tev(tsm,t,e_u)}{|M^-|}}{2}$$

for the following experiments.

8.1.2 Recognizing small terms

The first task we want to learn is the classification of terms into large and small terms. This is the classification problem corresponding most closely to the symbol counting evaluation heuristics used in theorem proving.

We selected a size limit of 10 symbols for the class split, i.e. terms with more than 10 symbols make up the positive class and terms with up to 10 symbols make up the negative class. This results in fairly even sized classes for both term set A and term set B. For term set A, there are 11069 positive terms and 8931 negative terms, resulting in a success rate of 50.57% for the random guesser and 55.345% for the naive learner. For term set B, we have 8252 positive terms and 11748 negative terms. In this case, the random guesser would achieve 51.52% and the naive learner 58.74%.

This test is very likely to be affected by the skewed distribution of terms noted in Section 8.1.1, as the classification criterion is directly linked to the same factor that limits the growth of the pseudo-random terms. The much higher density of generated terms for smaller sizes makes an over-specialization to these terms rewarding.

Table 8.2 shows the results achieved by the different term space mapping algorithms. In all cases, the best results are significantly better than both the random and the naive learner.

Recursive term space maps perform best for both term sets on this problem. As the problem is one of term topology, this is to be expected. Flat term space maps also perform quite good, although the size of the training set is very small compared to the training set size required in Theorem 6.3 for perfect classification. For very specialized index functions, we can observe *over-fitting* in both cases: The learned term space maps allow nearly no generalization to new examples, and the performance deteriorates towards those for a naive learner.

Index fct.	Term set A			Term set B		
	Flat	Recursive	Recurrent	Flat	Recursive	Recurrent
I_{ar}	67.1±0.95	74.5±0.97	61.5±1.24	75.0±1.18	76.0±1.34	58.9±1.26
I_{symb}	67.1±0.94	72.4±0.83	**62.5±1.34**	75.0±1.18	75.8±1.12	61.1±1.37
I_{id}	55.3±1.22	55.3±1.22	54.0±1.15	58.7±1.24	58.7±1.24	47.0±1.08
I_{top_1}	67.1±0.94	72.4±0.83	62.5±1.34	75.0±1.18	75.8±1.12	61.1±1.37
$I_{top'_1}$	67.2±0.93	72.7±0.96	61.9±1.17	75.1±1.17	76.4±1.20	**61.3±1.40**
I_{cstop_1}	67.2±0.93	72.7±0.96	61.9±1.17	75.1±1.17	76.4±1.20	**61.3±1.40**
I_{estop_1}	67.2±0.93	72.7±0.96	61.9±1.17	75.1±1.17	76.4±1.20	**61.3±1.40**
I_{top_2}	71.7±0.63	74.7±0.72	59.3±0.91	**78.8±0.55**	**80.8±0.63**	55.1±1.40
$I_{top'_2}$	**72.8±0.92**	75.2±0.76	57.4±1.07	77.9±0.75	79.8±0.77	54.1±1.52
I_{cstop_2}	72.7±0.88	**75.3±0.88**	57.1±1.08	77.6±0.92	79.2±0.87	53.9±1.50
I_{estop_2}	72.5±0.89	75.0±0.92	56.2±1.12	77.6±0.89	79.3±0.87	53.4±1.53
I_{top_3}	71.0±1.11	70.8±1.16	53.9±0.90	60.8±0.83	60.6±0.85	52.1±0.74
$I_{top'_3}$	70.5±1.40	70.3±1.36	53.9±0.80	60.2±1.11	60.1±1.12	51.4±1.08
I_{cstop_3}	70.2±1.51	70.0±1.44	53.6±0.67	60.2±1.08	59.9±1.06	50.9±0.96
I_{estop_3}	68.9±1.28	68.8±1.26	53.7±0.79	60.3±1.02	59.9±1.06	50.6±0.88
I_{top_4}	60.3±1.30	60.3±1.24	54.8±0.93	58.2±1.12	58.4±1.18	46.7±0.95
$I_{top'_4}$	60.0±1.29	60.0±1.23	54.8±0.96	58.3±1.12	58.5±1.16	46.7±0.97
I_{cstop_4}	59.5±1.37	59.5±1.33	54.7±1.01	58.4±1.12	58.5±1.14	46.6±1.10
I_{estop_4}	59.3±1.33	59.3±1.28	54.8±1.07	58.4±1.12	58.5±1.15	46.7±1.05
I_{top_5}	55.7±1.26	55.7±1.26	54.2±1.23	58.7±1.25	58.7±1.25	47.0±1.01
$I_{top'_5}$	55.6±1.26	55.6±1.26	54.2±1.23	58.7±1.25	58.7±1.25	47.0±1.04
I_{cstop_5}	55.6±1.27	55.6±1.27	54.2±1.22	58.7±1.25	58.7±1.25	46.9±1.05
I_{estop_5}	55.6±1.27	55.6±1.27	54.2±1.22	58.7±1.25	58.7±1.25	46.9±1.03
I_{opt}	67.1±0.95	72.4±0.58	61.5±1.24	75.0±1.18	76.0±1.33	58.9±1.26

Remarks: I_{opt} shows the result for the information-optimal term space maps. The best results for each term set and each type of term space map are marked in bold face.

Table 8.2: Results for term classification by size

Recurrent term space maps, finally, perform worst of all. This is unsurprising, as the flattening of the terms in the construction of recurrent term space maps destroys the very feature the classification depends on. For unsuitable index functions, the recurrent term space maps are even worse than the naive, and in some few cases even the random guesser.

The information-optimal term space maps show about average performance. For all cases except the recurrent term space map for term set B they are still significantly better than the naive learner. Information-optimal term space maps perform much better for the experiments described in the following sections. There are two reasons why they perform less well in this case. First, we try to separate a finite class (small terms) from an infinite class of terms. Consequently, the relative frequency of a class on the (finite) training set is *not* a good estimation for the overall probability of a term to belong to this class. As the relative information gain is based on the *entropy* of an estimated probability distribution,

this problem translates into a weakness of this selection criterion for index functions.

Secondly, we believe that this performance is influenced by the skewed term distribution as described above. This belief is supported by the fact that the information-optimal recurrent term space map for term set B performs relatively much worse then the corresponding term space map for set A. As set B contains repeated terms (nearly all of which are small in size), over-specialization to small terms pays off even more in this case.

To fix this problem, we would need to use a better estimation for the global probabilities of the classes. This estimation would need to take the probability distribution of terms in the test and training set into account.

8.1.3 Recognizing term properties

In this section, we test the performance of term space mapping for recognizing certain term properties. There are three different problems in this problem set:

- First, we try to recognize terms which carry certain symbols at certain positions. More exactly, we try to recognize terms t with the property that $1 \in O(t)$ and $head(t|_1) \in \{f_{01}, f_{02}, f_{03}, f_{04}, f_{11}, f_{12}, f_{13}\}$. This is the most specific constraint of this type we could find that still splits the two sets of terms into about equal parts. We refer to this problem as *Topstart*. For term set A, there are 9299 terms in the positive class and 10701 terms in the negative one. For term set B there are 10319 positive terms and 9681 negative ones. A random guesser would get 50.246% on A and 50.0509% on B, a naive learner 53.505% and 51.595%, respectively.

- The second problem, *Symmetry*, tries to recognize terms that are instances of $f_{21}(x, x)$ or $f_{22}(x, x)$. As only about 20 terms from the set A and B have this property, we constructed new test sets A' and B' for this problem. In both cases, every second term t from A or B was transformed into a symmetric term by randomly selecting a function symbol with arity 2 and using t at both argument positions. The resulting term sets are very well balanced. A' contains 10011 positive terms and 9985 negative ones, for random and naive success rates of 50.00006% and 50.055%, respectively. For B' we have 10015 positive and 9985 negative terms, with corresponding rates of 50.0001% and 50.075%

- The last problem we discuss in this section is *Symbol_occ*. The positive classes contains all terms in which the function symbol f_{01} occurs. For term set A this results in a 14514/5486 split (with random and naive success rates of 60.188% and 72.57%), for term set B we get a more even 12274/7726 split, and values of 52.586% and 61.37% for random guesser and naive learner.

Table 8.3 and Table 8.4 summarizes the results of the different experiments for the term sets A and B. We only give numbers for selected index functions (including the best index function) and the information-optimal index function.

Index function	Flat	Recursive	Recurrent
Topstart (Random: 50.246 Naive: 50.246%)			
I_{ar}	58.4±1.12	**100.0±0.00**	66.3±1.19
I_{symb}	58.4±1.12	99.9±0.06	66.3±1.14
I_{id}	53.5±1.08	53.5±1.08	60.0±1.16
$I_{top'_1}$	58.3±1.10	98.5±0.13	**66.7±1.15**
I_{top_2}	**99.2±0.16**	99.2±0.16	60.4±1.33
$I_{top'_2}$	98.0±0.40	98.0±0.40	59.1±1.36
I_{estop_3}	72.9±0.94	72.9±0.94	59.5±1.12
$I_{top'_4}$	60.3±0.96	60.3±0.96	63.4±0.93
I_{opt}	**99.2±0.16**	99.2±0.16	66.3±1.19
Symmetry (Random: 50.00006% Naive: 50.055%)			
I_{ar}	77.5±0.95	77.5±0.94	55.1±1.20
I_{symb}	77.5±0.95	77.5±0.94	54.5±1.05
I_{id}	49.3±0.81	49.3±0.81	53.8±14.14
$I_{top'_1}$	**100.0±0.02**	**100.0±0.02**	49.2±0.80
I_{top_2}	95.9±0.96	95.6±0.95	51.4±0.65
$I_{top'_2}$	97.6±1.66	97.6±1.66	50.7±0.95
I_{estop_3}	74.0±14.07	74.0±14.07	**61.1±1.26**
$I_{top'_4}$	55.9±5.79	55.9±5.79	56.5±9.84
I_{opt}	**100.0±0.02**	**100.0±0.02**	55.1±1.20
Symbol_occ (Random: 60.188% Naive: 72.57%)			
I_{ar}	61.3±1.06	63.5±1.05	62.1±1.44
I_{symb}	61.3±1.06	64.7±1.01	92.4±0.55
I_{id}	72.6±1.00	72.6±1.00	75.7±0.70
$I_{top'_1}$	61.4±1.05	65.3±0.92	**92.5±0.47**
I_{top_2}	61.3±1.21	69.3±1.27	89.9±0.86
$I_{top'_2}$	62.6±0.85	70.1±1.04	89.0±0.78
I_{estop_3}	71.8±0.84	72.4±0.87	75.9±0.84
$I_{top'_4}$	**73.1±0.82**	**73.1±0.83**	75.9±0.74
I_{opt}	72.6±1.00	72.6±1.00	92.4±0.55

Remarks: See Table 8.2.

Table 8.3: Term classification experiments (Term sets A/A')

As a general observation, we can see that both for the best and the information-optimal index functions the classification rates are significantly better than either the random guesser or the naive learner for nearly all term space map types and all experiments. As a second observation we can also note that the performance on term set B is usually better than on term set A (if compared to the respective rates of the naive learner). This indicates that the mapping algorithms do benefit from the ability to learn individual evaluations for

Index function	Flat	Recursive	Recurrent
Topstart (Random: 50.0509 Naive: 51.595%)			
I_{ar}	56.6±1.07	**100.0±0.00**	71.6±0.84
I_{symb}	56.6±1.07	**100.0±0.00**	71.9±0.82
I_{id}	60.3±1.01	60.3±1.01	54.3±1.63
$I_{top'_1}$	56.6±1.06	99.1±0.23	**72.3±0.73**
I_{top_2}	**99.7±0.15**	99.7±0.15	63.0±1.28
$I_{top'_2}$	97.7±0.43	97.7±0.43	63.0±1.48
I_{estop_3}	72.3±0.84	72.3±0.84	59.5±1.57
$I_{top'_4}$	61.1±0.99	61.1±0.99	54.4±1.45
I_{opt}	**99.7±0.15**	99.7±0.15	71.6±0.84
Symmetry (Random: 50.0001% Naive: 50.075%)			
I_{ar}	82.7±0.80	82.7±0.80	53.3±0.91
I_{symb}	82.7±0.80	82.7±0.80	53.1±1.05
I_{id}	62.0±0.70	62.0±0.70	55.2±9.29
$I_{top'_1}$	**100.0±0.00**	**100.0±0.00**	47.6±0.85
I_{top_2}	96.8±0.53	96.5±0.55	48.9±1.15
$I_{top'_2}$	97.8±1.19	97.8±1.19	48.2±0.88
I_{estop_3}	77.9±8.90	77.9±8.90	**58.5±1.45**
$I_{top'_4}$	66.0±3.44	66.0±3.44	56.5±6.69
I_{opt}	82.7±0.80	87.2±1.11	53.3±0.91
Symbol_occ (Random: 52.586% Naive: 61.37%)			
I_{ar}	63.9±1.22	71.7±1.47	60.7±0.97
I_{symb}	65.3±1.07	74.2±1.11	**92.1±0.50**
I_{id}	80.7±0.85	80.7±0.85	82.0±0.73
$I_{top'_1}$	65.3±1.06	74.1±0.98	91.7±0.51
I_{top_2}	71.9±1.08	77.9±0.55	88.8±0.42
$I_{top'_2}$	72.7±0.78	78.5±0.54	88.4±0.60
I_{estop_3}	78.5±1.00	80.2±0.83	81.7±0.64
$I_{top'_4}$	**81.0±0.66**	**81.1±0.61**	81.9±0.81
I_{opt}	80.7±0.85	80.7±0.85	**92.1±0.50**

Remarks: See Table 8.2.

Table 8.4: Term classification experiments (Term sets B/B')

repeated terms.

Let us now discuss the three individual experiments:

- For *Topstart*, both the flat and the recursive term space maps achieve perfect or nearly perfect results. As the distinguishing property is defined in terms of function symbols at absolute positions, this is as we expected from the theoretical discussion in Chapter 6. We can also note that the recursive term space map achieves a better

overall score with less specific index functions.

The information-optimal index function is the best index function for the flat case. It is not the best one for the recursive case, though. This shows a *general weakness* of the way index functions are selected in the recursive case. As the information-optimal index function is selected at each term position individually, unnecessarily specific index functions are selected at the top level in this case. The same index functions are used to split the example set for the recursive descent, and hence the relevant features are lost to the lower level basic term space maps. Section 9.3 discusses potential improvements.

Finally, the recurrent term space maps perform worst. As for the size-based features, the flattening of the terms destroys a part of the relevant properties (the absolute positioning of the function symbols), and hence no better performance is to be expected. The information-optimal index function is not the best one, but the differences are statistically insignificant in all cases.

- For the *Symmetry* problem, we achieve the absolutely best results for both flat and recursive term space maps for term set A. Both achieve perfect classification of the test set, and in both cases the information-optimal term space map achieves this performance as well. The abstraction defined by one of our of index functions is very well suited for this problem, and the entropy-based selection of index functions is able to identify the optimal function even though the classification performance *on the training set* is the same for the three index functions $I_{top'}$, I_{estop} and I_{cstop}.

  For term set B, we get the same best-case performance, but the relative information gain criterion fails to identify the best term space map. The reason for this is the particularly skewed term distribution. Since repeated terms are allowed in term set B, and since most generated terms are small terms, nearly all terms with arity 2 are artificially generated symmetrical case. Hence just checking the arity of the top function symbol results in nearly 90% classification correctness. The relative information gain criterion prefers this 4-way split with high accuracy (relative information gain 0.568412) to the much larger split with perfect accuracy it gets for $I_{top'_2}$, although only barely. The relative information gain for this split is 0.522210.

  The recurrent term space map again suffers from the fact that the absolute position of the relevant features is destroyed in flattening. It's performance and the best index function can be explained by the construction of our term sets A' and B'. Since most of the positive terms are constructed by combining a random top symbol and two copies of a term from sets A or B, the average *size* of positive terms is more than twice as large as the average size of negative terms. Hence terms matching a sufficiently large term top are likely to be positive, as are terms where different subterms occur more than once.

- The *Symbol_occ* problem shows the potential of the recurrent term space map if the classification-relevant property is not bound to an absolute term position. The

recurrent term space maps achieve the best classification results. For term set A, the information-optimal term space map performs insignificantly worse that the best one, for term set B the information-optimal recurrent term space map also is the one showing optimal performance.

The flat and recurrent term space maps are unable to recognize this class. Their performance on term set A is never significantly better than the naive learner, and the classification is often as bad as a random guess. For term set B, where memorization helps to classify small terms, the performance of flat and recurrent term space maps is better, and the information-optimal index functions again perform only insignificantly worse than the best index functions. Note that we discussed an equivalent classification problem on page 93 and pointed out that flat terms space maps cannot in general learn to recognize terms containing a certain function symbol.

8.1.4 Memorization

We have already noted that the term space maps generally seem to perform better on term set B, where they can utilize then fact that terms can occur in both training and test sets. In this section, we will test this ability further. We have randomly assigned classes to term from term set A, and have created a new (multi)-set of examples by taking two copies of each of these randomly classified terms. The task for the term space maps is to separate these two random classes again.

Please note that for 10-fold cross validation the chance for a term in the test set to also occur in the training set in this case is about 90% (90.0025 to be exact: there are two copies of each term, the remaining copy for an arbitrary term in the training set is either one of the 3999 terms in the test set or one of the 36000 terms in the training set). In other words, the best performance we can expect is about 95% (about 90% for the case of perfect memorization, and 5% by randomly guessing the class of the remaining 10% of terms). As the positive and the negative classes exactly of the same size, both the random guesser and the naive learner would only achieve 50%.

Table 8.5 shows the performance of our term space maps for this problem. As we can see, both the flat and the recurrent term space map achieve the optimal result of about 95%. In both cases the information-optimal term space map is I_{id}, as we can expect in a case where the classes are randomly selected and hence *no* term property carries any information about the classes. It is interesting to see that the recursive term space maps performs a lot better than the flat ones for all index functions but the identity function. The reason for this is, of course, that it tests more term properties than the single abstraction used by the flat term space map.

The recurrent term space map, on the other hand, does not perform significantly better than the random learner. This is easy to explain: Nearly every subterm will occur in both positive and negative terms. As the recurrent term space map evaluates all subterms against the same term space map, no clear evaluation can be expected.

Index function	Flat	Recursive	Recurrent
I_{ar}	50.0±0.59	52.2±0.67	50.0±0.71
I_{symb}	50.4±0.46	55.9±0.94	49.8±0.93
I_{id}	94.8±0.57	94.8±0.57	50.2±1.00
I_{top_1}	50.4±0.46	55.9±0.94	49.8±0.93
$I_{top'_1}$	50.4±0.46	56.6±0.97	49.8±0.96
I_{cstop_1}	50.4±0.46	56.6±0.97	49.8±0.96
I_{estop_1}	50.4±0.46	56.6±0.97	49.8±0.96
I_{top_2}	53.1±0.50	82.4±0.81	50.2±0.91
$I_{top'_2}$	55.2±0.58	84.3±0.69	50.1±0.96
I_{cstop_2}	55.3±0.54	85.1±0.87	50.1±0.93
I_{estop_2}	55.5±0.51	85.2±0.87	50.1±0.93
I_{top_3}	76.6±0.51	94.1±0.38	50.3±0.72
$I_{top'_3}$	78.7±0.72	94.2±0.53	50.3±0.86
I_{cstop_3}	79.4±0.66	94.3±0.51	50.3±0.83
I_{estop_3}	80.1±0.66	94.4±0.56	50.3±0.87
I_{top_4}	91.2±0.66	94.8±0.50	50.2±0.95
$I_{top'_4}$	91.6±0.69	94.8±0.52	50.2±0.94
I_{cstop_4}	91.9±0.71	94.8±0.55	50.2±0.96
I_{estop_4}	92.0±0.70	94.8±0.55	50.2±0.97
I_{top_5}	94.5±0.59	94.7±0.55	50.2±1.00
$I_{top'_5}$	94.5±0.59	94.7±0.55	50.2±1.00
I_{cstop_5}	94.6±0.59	94.7±0.56	50.2±1.00
I_{estop_5}	94.6±0.59	94.7±0.56	50.2±1.00
I_{opt}	94.8±0.57	94.8±0.57	49.8±0.93

Remarks: See Table 8.2.

Table 8.5: Term memorization with term space maps

8.1.5 Discussion

The experimental results in the previous sections demonstrated that different kinds of term space maps can learn different concepts. Flat and recursive term space maps are very good at learning concepts that can be described by localized term properties. Recursive term space maps usually generalize better to unknown examples. However, the automatic selection of information-optimal index functions does not usually lead to the optimal term space maps for the recursive case. For flat term space maps, on the other hand, the information-optimal term space map is usually among the term space maps that give the best performance.

Recurrent term space maps are good at learning non-localized term properties, a field of problems where flat and recursive term space maps perform bad both in theory and in practice.

Generally, the learning success is better if one of the possible index functions is suitable for a compact description of the concept to be learned. If this is the case, the relative information gain is a very good way to identify the optimal index function at least for flat and recurrent term space maps.

8.2 Search Control

In this section we describe the successes of term space mapping applied to the problem of learning search control knowledge in the E/TSM system described in the previous chapter.

For learning theorem provers, we are restricted to a finite set of proof problems from published collections. Moreover, the time for the evaluation of a search heuristic on a single problem is about 4-5 orders of magnitude larger than the time for the classification of an individual term. Cross validation is therefore neither practical nor customary for learning theorem provers.

As we are interested in the *increase* in the performance of our theorem prover, we select only *easy* problems as training examples. We test the resulting strategies on the complete TPTP, with special consideration for all *harder* problems. This split allows us to evaluate the generalization of the learned control knowledge to new, unknown proof problems. *Easy* and *hard* are defined with respect to the base strategy used to build the knowledge base. An easy problem is a problem that can be solved in less than 100 seconds by the base strategy, a hard problem is a problem that cannot be solved within this time limit.

We use all 3275 clause normal form examples from the TPTP problem library [SSY94] version 2.1.0 [SS97b] for our evaluation, and use two different knowledge bases. KB_1 contains 1251 proof experiences from unsatisfiable problems that can be solved in less than 100 seconds (including overhead for writing a protocol of the proof search) with the *RWeight* strategy described in Section 4.3. The second knowledge base, KB_2, contains 1427 proof experience that can be solved in less than 100 seconds with the *RWeight/FIFO* strategy, the strongest of the strategies analyzed in Section 4.3. In both cases we only included proof experiences, not experiences for problems which can be shown to be satisfiable (by saturating them without deriving the empty clause). For both knowledge bases we selected all proof clauses and up to the same number of clauses close to the proof to represent each proof search.

Table 8.6 shows some data about the two knowledge bases. There are two interesting observations about this data:

- First, the number of clause patterns is much lower than the number of original clauses. Moreover, while KB_2 contains significantly more proof experiences and clauses, the number of clause patterns does not increase proportionally. Both of these facts indicate that the represented proofs share a lot of clauses with the same structure.

- Secondly, we can see that the memory requirements are fairly moderate by current standards. The largest part of the knowledge base is taken up by the proof experience archive. The functional part is small enough to allow e.g. integration of this part into

	KB_1	KB_2
Proof experiences	1251	1427
Generated with strategy	RWeight	RWeight/FIFO
Original clauses	54653	61813
Distinct clause patterns	12474	12807
Size (with archive) in kilobyte	16048	18241
Size (functional part only) in kilobyte	4047	4634

Table 8.6: Knowledge bases used for the evaluation of E/TSM

a program (to speed up processing times and reduce overhead) or easy distribution to potential users. The storage space taken up by the knowledge base do not seriously restrict the deployment of the prover in any reasonable environment.

In the following section, we will present the results obtained by E/TSM with the two different knowledge bases and a variety of TSM-based search heuristics. In all cases we used the standard KBO (see A.1.3) and the literal selection function `SelectLargestNegLit` (see A.2.2), exactly as we did for the proof experience generation.

All results have been obtained in compliance with the guidelines for use of the TPTP. TPTP input files were unchanged except for removal of equality axioms and syntax transformation. The performance was evaluated on a cluster of SUN Ultra 60 workstations running at 300 MHz. CPU time per attempt was limited to 300 seconds, memory to 192 MB.

8.2.1 General observations

All of the TSM-based strategies rely on a fairly large set of parameters. For convenience, we will give a short overview here.

- There are 7 different variables for the initial evaluation of clause patterns:
  - w_{pn} is the relative weight given to the number of proofs a pattern occurred in.
  - w_{pd} is the relative weight given to the average proof distance of a pattern.
  - w_{mp} is the relative weight for the number of modifying inferences with clauses matching the pattern and that contributed to a proof.
  - Similarly, w_{mn} is the relative weight for the number of modifying inferences not contributing to a proof.
  - w_{gp} is the relative weight for the number of generating inferences contributing to a proof.
  - w_{gn} is the relative weight for the number of superfluous inferences contributing to a proof.

  - w_{sc} is the relative weight for the number of subsumption inferences.

- There are parameters controlling the number of similar proof experiences that are selected to guide a new problem:

  - *sel_abs* gives the absolute maximum number of proof experiences to select.
  - *sel_rel* gives the maximum number of proof experiences to select as a fraction of the number of all proof experiences.
  - *sel_dist* gives a relative limit for the maximum distance for a proof experience from the new problem.

- There is the TSM type (*flat, recursive or recurrent*), the index function, and the clause pattern type (*flat* or *recursive*).

- Finally, there are the parameters for the final evaluation:

  - w_{learn} is the influence of the TSM-evaluation.
  - w_f and w_v are the variable and function symbol weights for the underlying clause weight strategy.

This set of parameters results in a very large number of possible strategies. Due to the size of the parameter space and the limited computing resources we have performed only a cursory survey so far. We will only present some selected results and give an overview of our other findings here. Full protocols of all test runs are archived and can be obtained upon request.

- Of the seven weights for the a-priory evaluation, only the first two (number of proofs and average proof distance) have a strong individual influence on the performance of the resulting strategy, with the proof distance being very slightly more useful. Effects of the other parameters are not significant. For the test results presented later we use $w_{pn} = -20$, $w_{pd} = 20$, $w_{mp} = -2$, $w_{mn} = -1$, $w_{gp} = 0$, $w_{gn} = 1$, $w_{sc} = -1$. Keep in mind that a positive weight means that the evaluation of the clause becomes worse with the corresponding parameter, a negative value implies that the evaluation becomes better if the parameter value increases.

- The performance of the learning strategies is fairly stable for values of w_{learn} between 0.5 and 20. It drops off rapidly for smaller values and slowly for larger values. Unless otherwise mentioned, we use $w_{learn} = 5$ for the following results.

- The selection of similar proof experiences is best guided by *sel_dist*. The best results were obtained for *sel_dist* = 1, i.e. for selecting all problems that have a less than average distance to the current proof problems. We have always used values for *sel_abs* and *sel_rel* that do not influence the selection in the following results.

- The information-optimal index function is almost invariable I_{id} for flat and recurrent term space maps. We have therefore used fixed index functions only.

We have kept $w_f = 2$ and $w_v = 1$ fixed for the presented results. This allows us to compare the performance of the learning strategies with the base strategies used to build the knowledge base as well as with the standard clause weight evaluation. We always used the default recursive clause pattern encoding for flat term space maps and the flat clause pattern encoding for recursive and recurrent term space maps.

8.2.2 Performance with KB_1

Table 8.7 shows the comparative performance of a set of learning and non-learning strategies.

As a first observation, we can see that among the homogeneous strategies (those not interleaved with a first-in/first-out component) all of the TSM-based strategies outperform the non-learning ones significantly. Particularly among the hard problems, the learning strategies can solve between 50% and 250% more problems than the conventional ones.

Similarly, among the strategies that interleave a base strategy and FIFO, all of the learning strategies are again significantly better than the conventional one. In this case, the improvements are split between better performance on the training examples and on the other problems. The best learning strategy can solve 24 training problems and 29 hard problems that the strongest conventional heuristic, RWeight/FIFO, cannot solve at all.

The computational overhead of the learning strategies is notable. If they are not allowed to profit from their additional effort (as in the case of the TSM_0-strategies), they perform worse than the Weight_2 heuristic, although they generate the same proof derivations.

Finally, we can see that all strategies can show for nearly the same number of formulae that they are not unsatisfiable, but have a model. Even the generally very weak pure FIFO strategy can prove this property for 87 problems. This suggests that most of these problems in the TPTP are very easy, and are proven by nearly every strategy.

Let us now discuss some of individual learning strategies in more detail:

- Both of the learning strategies using flat term space maps can reproduce nearly all of the training examples. There is no significant difference between the heuristic with and without proof experience selection. The probable reason for this is the low degree of generalization provided by this particular choice of term space map. Only patterns from the knowledge base that exactly match a new clause are used in the evaluation of this clause. But clauses from very different proof problems are unlikely to have this property. Therefore, a selection of suitable knowledge is performed automatically within the TSM.

- The heuristics using recursive term space maps perform slightly worse. Nevertheless, they still improve the overall performance very significantly if compared to both Weight_2 and RWeight. In this case, the selection of similar experiences does result

Strategy	Training Problems	Other Problems		Overall Successes
		Proofs	Models	
Weight_2	1201	95	87	1383
RWeight	1251	68	87	1406
RWeight/FIFO	1218	259	88	1565
TSM(flat,I_{id},all)	1249	149	87	1485
TSM(flat,I_{id},sel.)	1249	149	87	1485
TSM(recursive,I_{symb},all)	1243	146	87	1476
TSM(recursive,I_{symb},sel.)	1242	155	87	1484
TSM(recurrent,I_{id},all)	1242	169	88	1499
TSM(recurrent,I_{id},sel.)	1247	169	88	1504
TSM(flat,I_{id},all)/FIFO	1250	262	86	1598
TSM(recursive,I_{symb},sel.)/FIFO	1219	286	87	1592
TSM(recurrent,I_{id},sel.)/FIFO	1232	288	88	1608
TSM_0(flat,I_{id},all)	1197	86	87	1370
TSM_0(recursive,I_{id},all)	1191	82	87	1360
TSM_0(recurrent,I_{id},all)	1190	82	87	1359

Remarks: Shown are the number of successfully terminated proof searches within a time limit of 300 seconds CPU time on a SUN Ultra 60/300 workstation. TSM-based strategies are described as *TSM(tsm_type, index_fct, selection_state)*, where *tsm_type* gives the type of the TSM, *index_fct* describes the index function used in the TSM and *selection_state* describes if all proof experiences from KB_1 were used or is only similar ones (with $sel\_dist = 1$) were selected. For the other parameters see section 8.2.1. Strategies of the type *Base/FIFO* select 5 out of every 6 clauses according to the base heuristic, the last one according to the FIFO strategy. The TSM_0-strategies use the same clause selection as Weight_2, but simulate the overhead of the corresponding TSM evaluation functions. For a more detailed discussion of the overhead see Section 8.2.4.

Table 8.7: Performance of learning strategies with KB_1

in an improvement in the learning heuristics. As recursive term space maps, especially with a very general index function as I_{symb}, generalize better than flat ones to unknown examples, this is consistent with our analysis for the flat case above.

- Finally, the recurrent term space maps show the best performance. Both perform very well on the hard problems, and especially the heuristic with experience selection reproduces nearly all of the training problems. Recurrent term space maps allow generalization to unknown examples even with I_{id} as index function, as all subterms are evaluated individually. In this case, particular subterms seem to play important roles in the proof process, and are recognized as such.

In the analysis of the RWeight/FIFO strategy in Section 4.3, we noted that in some proof problems the clauses describing the theorem to be proved are rather large, and

Strategy	Training Problems	Other Problems		Overall Successes
		Proofs	Models	
RWeight/FIFO	1427	50	88	1565
TSM(flat,I_{id},all)	1422	113	86	1621
TSM(recursive,I_{symb},sel.)	1394	94	86	1574
TSM(recurrent,I_{id},sel.)	1417	120	88	1625
TSM(flat,I_{id},all)/FIFO	1425	141	87	1653
TSM(recursive,I_{symb},sel.)/FIFO	1400	121	87	1608
TSM(recurrent,I_{id},sel.)/FIFO	1419	132	88	1639
TSM(flat,I_{id},all)/FIFO 8/1	1425	158	87	1670

Remarks: See Table 8.7. The TSM-based heuristics now use KB_2.

Table 8.8: Performance of learning strategies with KB_2

hence are selected very late by symbol-counting heuristics. However, such goal clauses typically change from problem to problem, while our learning strategies will primarily learn knowledge about parts of the proof search that are common to many proof problems. For this reason, we have also interleaved some of the learning strategies with FIFO. The success justifies this decision. The resulting strategies are much stronger than both the non-learning and the homogeneous learning strategies.

As for the homogeneous case, performance of the flat term space map is better than for the recursive term space map, and the recurrent map performs best.

8.2.3 Performance with KB_2

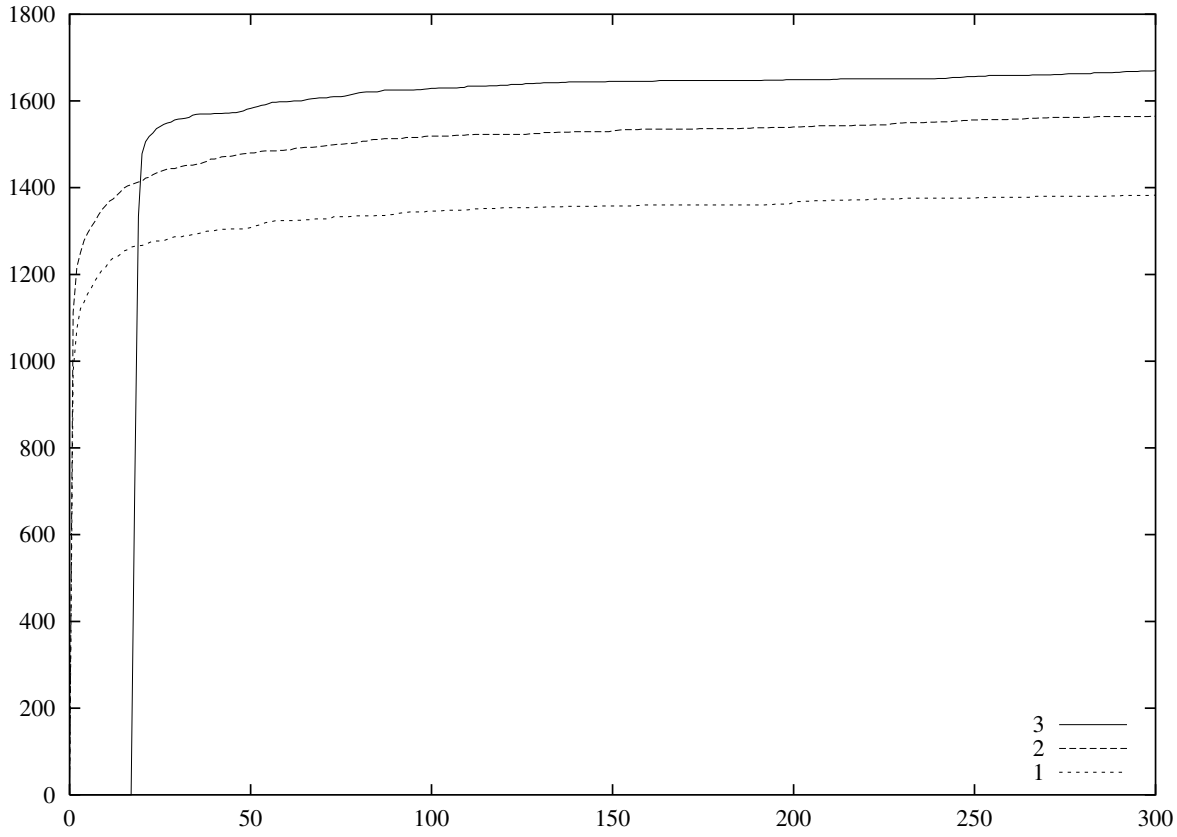
As we saw in the previous section, learning on the examples easily solvable with RWeight lead to significant improvements over all non-learning strategies. However, it required interleaving with FIFO to make learning strategies better than the best non-learning strategy. We will now use the larger knowledge base KB_2 constructed from problems that are easy for RWeight/FIFO. Table 8.8 shows the results. We only tested those evaluation heuristics from each class that performed particularly well on the smaller knowledge base.

We can see that in this case the learning heuristics perform better, both overall and particularly on hard problems, than the RWeight/FIFO strategy (and, by extension, all of the non-learning heuristics). Even without interleaving FIFO, the improvements are quite significant for the heuristics using flat and recurrent term space maps. If we again interleave FIFO, the performance increases still further. The improvement, however, is less significant than in the case of the smaller knowledge base. This is not surprising, as many problems solvable only with interleaved FIFO are already among the training examples, and can once again be reproduced in nearly all cases.

It is interesting to see that that in this case the the pure recurrent strategy is only slightly better than the pure flat strategy, and that for the interleaved case this relation is inverted. This is compatible with earlier results for pure pattern memorization in the

unit-equational case, where we found that the performance of pure pattern-based strategies improves continuously with the number of proof experiences [DS96a, DS98].

Table 8.8 also contains data about the best TSM-based strategy we have found so far. As the evaluation of clauses with term space maps is a lot better than for standard symbol counting, we have reduced the *pick-given ratio* to 8-1, i.e. we select 8 out of every 9 clauses with *TSM(flat,I_{id},all)* and the last one with the first-in/first-out strategy.



Remarks: Shown is the number of successes over runtime (in seconds). Plotted results are for:
1: Weight_2
2: RWeight/FIFO
3: TSM(flat,I_{id},all)/FIFO 8/1 (based on KB_2)

Figure 8.2: Comparion of learning and non-learning strategies

Figure 8.2 compares Weight_2 (the base strategy modified by the term space maps), RWeight/FIFO and the best TSM strategy in more detail. It shows the number of successes a strategy achieves over the run time limit for a proof attempt. We can see that the conventional strategies find many proofs during the constant overhead time of the learning

heuristic, but that they are nearly immediately overtaken once this strategy starts the main inference process. From about 50 seconds to 300 seconds the three plots run more or less parallel, which indicates that the advantage of the learning heuristic should be stable even for longer run-times.

Finally, Table 8.9 compares the three strategies on different types of proof problems. We contrast the performance of the strategies for 6 different classes of formulae: Unit formulae, Horn Formulae and general formulae, with and without equality.

Problem Type	No. in Class	$Weight_2$		RWeight/FIFO		TSM/FIFO 8/1	
		Proofs	Models	Proofs	Models	Proofs	Models
Unit, no eq.	11	8	3	8	3	8	3
Unit, eq.	402	240	1	258	1	275	1
Horn, no eq.	557	384	5	417	5	443	5
Horn, eq.	321	172	2	219	3	222	3
General, no eq.	766	211	73	238	73	247	72
General, eq.	1218	281	3	337	3	388	3

Remarks: Shown are successes within the standard time limit of 300 seconds.

Table 8.9: Performance comparison for different problem types

We can see that the learning strategy improves the performance of the prover for all problem classes except the trivial case of unit problems without equality. However, the improvement for general formulae with equality is particularly impressive. This is both the largest and the most difficult problem class in the TPTP. It contains a very large group of 886 hard problems based on a common axiomatization of set theory and described in Appendix B.6. The learning strategies apparently are able to profit from this large group – they can prove 198 of these problems (as opposed to 160 for RWeight/FIFO and 145 for $Weight_2$).

8.2.4 Overhead

We will now take a more detailed look at the computational overhead of the learning strategies compared to pure symbol counting and compared to strategies using term orderings (see Section 4.3 and Appendix A.2.1). We can compare this overhead very well, as we can select parameters for all strategies that lead to very nearly the same proof derivation in all cases. For the TSM-based strategies we simply set the value w_{learn} to 0, for the ordering-based strategy we set the weight multipliers for maximal terms and literals to 1.

For this measurement, we will again consider our six standard examples described in Appendix B. We consider these problems to be fairly representative for problems solvable by E with conventional strategies. The set includes unit, Horn and general problems, and also includes pure equality problems, problems with some equality, and a problem without equality.

We compare two conventional search heuristics and four different TSM-based strategies that use term space maps similar to those of the best learning heuristics above:

Weight$_2$: This heuristic is described in Section 4.3. It uses the clause weight with $w_f = 2, w_v = 1$ as an evaluation.

RWeight$_0$: This heuristic determines maximal terms and literals in a clause, ignores the result and returns the same result as the previous one.

TSM(flat,I_{id},all): This strategy uses flat term space maps with index function I_{id}. It selects all problems from the knowledge base KB_1 and evaluates clause patterns against the resulting TSM. However, it ignores the result and returns only the term weight as above.

TSM(flat,I_{id},sel.): This search heuristic is nearly identical to the previous one, but only selects proof experiences that with a distance smaller than the average distance of all problems from the knowledge base.

TSM(recursive,I_{symb},all): This variant uses recursive term space maps with the index function I_{symb}. Otherwise it is identical to the first TSM-based heuristic.

TSM(recurrent,I_{id},all): Again, this heuristic is similar to the first one. The only difference is the use of a recurrent term space map.

All TSM-based heuristics use KB_1. Tables 8.10 and 8.11 show the times for different parts of the prover run on our 6 test problems. We can see that for the conventional strategies the startup times (time from the start of the program to the first inference) is insignificant. For the learning strategy, the total startup time varies from about 10 seconds to about 45 seconds. It consists of a very nearly constant time of about 7 seconds for the selection phase and a variable part for the term space mapping phase. This second part depends on both the number of proof problems selected and on the term space map type.

- For flat term space map, the mapping time for all 12474 distinct clause patterns in the knowledge base is about 9 seconds. If we restrict the selection of problems to those that are similar to the current proof problem, this time drops to about 2–4.5 seconds, depending on the problem.

- For recursive term space maps, the mapping time relatively moderate. Mapping all 12474 clause patterns takes only about 4 seconds, despite the fact that 8675 distinct term space maps are created. The main reason for this is that the index function I_{symb} is very cheap to compute and does not require any complex operations or term copying. If we use I_{id} here (which is uninteresting, as in this case the resulting strategy is identical to the one resulting from a flat term space map), mapping time rises to nearly 40 seconds.

Strategy	Startup time		Inference	Overall
	Selection	TSM	Time	Time
INVCOM				
Weight_2	0.030		-	0.030
RWeight_0	0.030		-	0.050
TSM(flat,I_{id},all)	6.870	9.280	-	16.040
TSM(flat,I_{id},sel.)	6.870	3.890	-	10.820
TSM(recursive,I_{symb},all)	6.930	3.750	-	10.830
TSM(recurrent,I_{id},all)	6.860	24.750	-	31.610
BOO007-2				
Weight_2	0.040		36.740	36.780
RWeight_0	0.030		37.660	37.690
TSM(flat,I_{id},all)	6.980	9.000	48.240	64.220
TSM(flat,I_{id},sel.)	6.970	2.570	47.770	57.310
TSM(recursive,I_{symb},all)	7.060	3.960	44.860	55.880
TSM(recurrent,I_{id},all)	6.990	24.850	57.510	89.350
LUSK6				
Weight_2	0.030		16.260	16.290
RWeight_0	0.030		16.840	16.870
TSM(flat,I_{id},all)	6.860	9.200	23.350	39.410
TSM(flat,I_{id},sel.)	6.860	2.870	23.540	33.270
TSM(recursive,I_{symb},all)	6.930	3.940	21.540	32.410
TSM(recurrent,I_{id},all)	6.880	24.790	30.670	62.340

Remarks: Shown are CPU times in seconds on a SUN Ultra 60/300 as returned by the UNIX operating system timer. The accuracy of this timer is limited, we have observed differences of up to ± 0.02 seconds for different runs on the same task. A dash implies that the time is negligible and cannot be measured with any accuracy. *Selection* is the time for the selection of similar proof problems, *TSM* is the time for the construction of the term space map. Non-learning strategies only have an overall startup time. Inference time is the time actually spend processing clauses.

Table 8.10: Startup and inference time comparison

- Finally, for recurrent term space maps we have by far the largest mapping time. While only a single term space map is created, the number of terms is increased very significantly by the flattening procedure that represent a term as the set of all of its subterms.

In addition to the constant overhead, the evaluation of new clauses also has a per-clause overhead resulting from pattern-transformation and evaluation. If we look at the times taken for the inference process, we can estimate this part very well.

- First, the performance of the two conventional strategies is very similar. The more

Strategy	Startup time		Inference	Overall
	Selection	TSM	Time	Time
HEN011-3				
Weight_2	0.030		44.330	44.390
RWeight_0	0.030		44.430	44.490
TSM(flat,I_{id},all)	6.950	9.040	52.990	68.980
TSM(flat,I_{id},sel.)	6.970	4.570	52.870	64.410
TSM(recursive,I_{symb},all)	7.080	3.960	51.670	62.710
TSM(recurrent,I_{id},all)	6.980	24.830	59.460	91.270
PUZ031-1				
Weight_2	0.030		0.040	0.070
RWeight_0	0.030		0.050	0.080
TSM(flat,I_{id},all)	7.040	9.090	0.040	16.170
TSM(flat,I_{id},sel.)	7.110	4.090	0.040	11.240
TSM(recursive,I_{symb},all)	7.180	3.560	0.050	10.790
TSM(recurrent,I_{id},all)	7.070	24.740	0.030	31.840
SET103-6				
Weight_2	0.080		40.430	40.510
RWeight_0	0.090		41.020	41.110
TSM(flat,I_{id},all)	6.880	9.080	53.940	69.900
TSM(flat,I_{id},sel.)	6.890	2.340	52.480	61.710
TSM(recursive,I_{symb},all)	6.960	3.990	50.350	61.300
TSM(recurrent,I_{id},all)	6.880	24.890	66.490	98.260

Remarks: See table 8.11.

Table 8.11: Startup and inference time comparison (continued)

complex ordering-based approach is slightly slower, however this effect is near the limit of measurability.

- All of the learning strategies are significantly slower than the conventional ones. The difference in inference speed varies between 20% and 60%.
- The two flat search strategies perform very similar, despite the different size of the term space maps used. Our index functions are realized using splay trees with an average retrieval time that is logarithmic in the number of entries, so that only a small increase is expected. However, this is also an indication that the largest part of the extra work for the learning strategies with flat evaluation is the generation of the *representative pattern* for new clauses.
- The recursive evaluation is, in general, slightly faster than the flat evaluation. This is unsurprising, since the recursive evaluation requires only one term traversal (with the very cheap index function I_{symb}, while the flat evaluation requires a search in

the index set, which usually involves multiple full term comparisons. Again, the similarity of the overhead is an indication that the evaluation cost is dominated by pattern computation.

- Finally, the recurrent evaluation is again the most expensive one. All subterms in a pattern have to be evaluated against a very large term space map. This results in an overhead that is about twice as large as for flat evaluation.

All in all we find that the overhead for the complex learning strategies is significant, but acceptable. The fact that the learning strategies perform as good as they do on the complete TPTP shows that the work for intelligent clause evaluation is well-spent.

8.2.5 Discussion

We have seen that the learning strategies significantly improve the performance of the proof system. They always perform better than the base strategy used to generate training examples. This is particularly significant as the learning strategies operate with a fairly large computational overhead. The fact that they perform as well as they do despite this overhead is an indication that our approach is indeed able to learn useful search control knowledge.

This is a justification for our model of learning. Apparently, both the representation of proof experiences as sets of clauses that are close to the proof in the proof derivation graph and the generalization of clauses into representative patterns is an adequate abstraction of the overall search process and retains sufficient information to learn good search decisions.

Similarly, the fact that the selection of similar proof experiences usually improves the performance of a learning heuristic indicates that the used criterion of similarity is adequate and can identify good proof examples at least among those problems present in the TPTP problem library.

The fact that the best results were obtained with flat and recurrent term space maps and the I_{id} index function are an indication that the strategies primarily learn *domain knowledge*, i.e. knowledge about important facts and objects in the modeled domains as opposed to calculus-specific technical knowledge. This is supported by the fact that none of the less detailed term abstractions gave a better relative information gain than the term identity function.

We have once more shown that effort into good guidance for the inference process of a theorem prover is usually well invested. This holds for both the effort of the researcher as well as for the computational cost associated with powerful search guiding heuristics.

8.3 Summary

Our experimental results have demonstrated a variety of important points both about term space mapping and about our approach to the learning of search control knowledge. The most important results about term space mapping are listed below:

- Term space mapping can be used to recognize a variety of non-obvious term properties from example sets. The learning success increases if the selected index function is appropriate for representing the necessary concepts compactly.
- Flat and recursive term space maps are good for learning localized properties of terms. Recursive term space maps typically learn the same concepts with a more general index function (i.e. a stronger abstraction).
- Recurrent term space maps are good for learning non-localized properties of terms.
- The relative information gain is a very good measure for the quality of different index functions.

Results about the learning of search control knowledge include the following items:

- The learning system E/TSM outperforms the same prover with conventional search heuristics significantly. This validates the decisions we made in the design of the proof system.
- In all tested cases, the learning heuristic performs much better on hard problems than the conventional strategy that has been used to generate training experiences for it.
- This increase in performance also holds if both conventional and learning heuristics are interleaved with a first-in/first-out strategy for clause selection.
- Best results are achieved for the flat and recurrent term space maps with the identity index functions. However, recursive term space maps with I_{symb} also lead to an improved performance.
- The successes of the learning heuristics are achieved despite the significant computational overhead for the clause evaluation that that slows down the inference process.
- The selection of similar training examples can improve the performance of the prover further. The importance of this feature seems to become larger for term space mapping variants that are better at generalization to unknown terms.
- There are some indications that our current system learns primarily *domain knowledge*, not calculus-specific knowledge.

Chapter 9

Future Work

The previous chapter has demonstrated the success of our prototypical implementation of a learning theorem prover. We have layed both the theoretical and the practical foundations for a capable, fully automatic, proof system that can be adapted to different domains and that can profit from previous experiences especially for the solution of hard problems.

However, we can still significantly improve both our proof system and the underlying ideas. In this chapter we will discuss some of the options for future work. There are tree main fields: Improving the practical usefulness of the proof system by improving the various interfaces to the human user and particularly to other reasoning systems, improving the efficiency of the implementation of our existing learning techniques, and finally improving the expressive power of *term space mapping*.

9.1 Proof Analysis

At the moment, the analysis of a proof search and the selection of representative clauses is based on a very lean, specialized protocol of the inference process. This protocol is not very suitable for other tasks. For the future, we plan to use a more general, prover-independent protocol, similar to PCL [DS94a, DS94b, DS96b].

Based on this general format, we will implement a variety of tools to complement our prover. Among these tools we want to implement a proof checker, a proof structuring tool and a proof presentation program that transforms superposition proofs into a human-readable format.

A more general format also will allow us to implement a more detailed proof analysis. This would in particular allow us to try to learn good literal selection functions as mentioned in 4.2.4.

A major advantage of a general proof search communication language is the potential ability to exchange proof experiences from different provers. It is well known that even provers implementing very similar calculi often can prove very different problem sets. The ability to transfer search control knowledge between provers therefore may lead to significant synergy effects.

9.2 Knowledge Selection and Representation

At the moment, E/TSM represents individual proof problems with a feature vector and a set of annotations in the clause pattern set. We only use feature vectors for determining similarity of proof problems. This approach works fairly well for the TPTP, however, as already discussed in Section 5.1 and Section 6.1, the selection of a feature set strongly limits the concepts of similarity that can be expressed. In [Bra98, SB99] we developed similarity measures based on recursive term space maps generated from the problem formulae. These similarity measures have already been implemented for DISCOUNT/TSM, and seem to complement the feature-based approaches well. We will transfer this approach to E/TSM to further improve the selection.

A core concept of our knowledge representation is the *representative clause pattern*. We use such clause patterns to represent potentially large numbers of equivalent clauses or clauses with a similar structure by a unique element. As we need to transform all newly generated clauses into their respective representative pattern, the efficiency of this operation is fairly important for the total efficiency of the prover with learning strategies.

The algorithm introduced in Section 5.2 is fairly straightforward. It certainly can be improved, in particularly for the worst case. Some ideas are based on observations of this worst case:

- Consider a clause of the form $f_1(x_1) \simeq g_1(y_1) \vee f_2(x_2) \simeq g_2(y_2) \vee f_n(x_n) \simeq g_n(y_n)$. All terms and literals contain disjoint sets of symbols, and all terms have exactly the same structure. Therefore all literals and literal encodings are equivalent under the ordering $>_{preord}$ or any other ordering that is compatible with function symbol renaming, and such orderings do not constrain the search for the representative clause pattern. However, in this particular case, *all* term encodings of the clause lead to the *same* representative clause pattern, and hence no such search is necessary.

  More generally, whenever we have a partially constructed clause pattern and the remaining literals have the same structure, but do not share any unrenamed symbols, the order of these literals in the clause encoding does not influence the resulting clause patterns.

- If, on the other hand, we have a set of structurally identical literals that do share some unrenamed symbols, we can use the position of these symbols to constrain the possible literal orderings.

  Consider the case of the clause $a \simeq a \vee a \simeq b \vee c \simeq c$. In this case, we can directly compute the representative pattern, without any search at all. Only the first and the third literal can be minimal in the final pattern representation, as the second literal contains an additional function symbol that immediately would make this literal and hence the resulting clause pattern larger. However, while both literals have equivalent pattern representations, the repetition of the function symbol a in the second literal forces the ordering of literals in the minimal pattern to be the same as in the original clause.

We believe that such relationships can be used to combine our current algorithm with explicit constraints to speed up the pattern computation for large clauses.

9.3 Term-Based Learning Algorithms

Term space mapping, as currently described, can be refined in a variety of ways.

At the moment, we select a single homogeneous index function from a small set of possible function to partition the term space. However, Theorem 6.2 gives us a lot of freedom for creating new index functions by combining existing ones. In particular, as any term top function is an index function, we can select arbitrary term tops to select a subset of terms.

One possible way to use this is to systematically search for good term space alternatives in a way analogous to the method used in the CN2-algorithm [CN89]. In this case, individual hypotheses (corresponding to term space alternatives) can be evaluated for significance (number of matching examples) and correctness (percentage of examples in one class). Possible alternatives can be described not only by term top variations, but also by term feature collections similar to those used in *path indexing* [McC92], or by combinations of term-based and numerical index functions.

Finally, to overcome the problems of recursive term space maps with information-optimal index functions, we can use two different partitions of the term space, a maximally general one (taking only the term geometry into account) and an arbitrary different one to induce the evaluations.

Another alternative is to replace term space maps completely. We have performed some preliminary experiments for the application of folding architecture networks in saturating theorem provers [SKG97]. However, there is no current implementation of a saturating theorem prover with a folding architecture network component for search control. We consider the combination of the signature-abstracting properties of patterns and the expressive power of folding architecture networks to be particularly interesting. As the learning times for folding architecture networks are very high, learning on demand as implemented at the moment is probably impossible. However, we can pre-train many networks on classes of problems and use the selection module to find a suitable network for new proof problems.

9.4 Domain Engineering and Applications

The last years have seen a constant and sometimes dramatic improvement in the power of automated theorem provers. We believe that this will lead to a strong increase in the practical application of theorem provers and related technologies.

As we described in the introduction to this thesis, theorem prover are already being used for many important tasks. In most of these cases the theorem prover is used in a given application domain, which can be encoded using a standard signature and a (possibly hierarchic) axiomatization. It is very likely that future applications will follow this pattern.

As we have seen in the previous chapter, our approach seems to be particularly useful for capturing (general) domain knowledge. A very interesting application of our theorem prover is therefore the *modeling* of one or more individual domains. In this case, special function symbols that describe e.g. unusual concepts in the application can be exempt from the generalization in the representative patterns and thus receive special treatment. This would have the double effect of speeding up pattern computation (as fewer potential patterns have to be explored) and of representing special knowledge (that may even come from non-standard proof searches) about these symbols and the sub-domains in which they play an important role.

If this approach is successful, a next step would be the automatic detection of such application domains, using techniques as described in [DK96] and [HJL99], and the automatic switch to one of several knowledge bases. Alternatively, if only a small number of domains is involved, it might be possible to combine this knowledge in a single knowledge base.

9.5 Other Work

There are some additional assorted avenues for future work:

- Up to now, we only make very limited use of meta-knowledge in E. To further improve the performance of the prover in default mode, we want to use meta-learning to learn good term orderings (e.g. by automating the process used to optimize the Waldmeister theorem prover) and to select good literal selection functions.

- We do need to do further evaluation of E/TSM. Important areas are larger knowledge bases, different literal selection functions and term orderings, and different weights for the clause weight heuristic modified by the term space map. Based on the result of these additional tests, we will integrate learning strategies into the automatic mode of the prover.

- In the future, we want to further integrate E and SETHEO into a tightly coupled system E-SETHEO, where E is used as a bottom-up saturating system and SETHEO tries to find top-down proofs for pre-saturated formulae, using some variant of the METOP calculus [Mos96]. As SETHEO cannot cope well with too many clauses, the selection of a good subset of clauses is crucial for the performance of the combined system. This task is very similar to the clause selection within E, however, learning good decisions for this choice point requires the analysis of combined proofs.

- For many applications of theorem provers, explicit and human-readable proofs are either a necessity or at least strongly desirable. Especially for a combined system proof presentation requires techniques closely related to the proof analysis necessary. We therefore expect to deal with both of these problems at once by finding compatible and general standard representations for both top-down and saturating proof searches.

- Finally, while the code for the learning search heuristics has been tested extensively and works flawlessly, we believe that with the experiences gained from our first implementation we can now create a much more compact, structured, and efficient implementation of the same concepts.

Chapter 10

Conclusion

In this work we have for the first time developed an automated theorem prover for clausal logic that combines solutions for all phases of the learning cycle. Our approach also is the first attempt to learn search control knowledge for a saturating theorem prover for full clausal logic.

To achieve this goal, we have generalized a variety of techniques previously only applied to the simpler unit-equational case. This includes the introduction of numerical distance measures to determine similarity between proof problems, and the proof analysis techniques necessary to represent proof derivations by a small set of annotated clauses standing for important search decisions. A particularly important contribution is the introduction of representative clause patterns, as a generalization of the representative term patterns we developed earlier. Representative patterns allow us to abstract from the usually arbitrarily chosen symbols in different proof problems.

We also developed *learning by term space mapping*, a class of learning algorithms that learn evaluations for terms by partitioning the set of all terms in different ways, and by associating evaluations from a representative term set with each partition. Term space maps can incorporate a very wide variety of different term abstractions. To select the best of these abstractions, we have developed the information-theory based concept of *relative information gain* that compares the useful information gain induced by an abstraction to the unnecessary information cost for applying it. Experimental results show that this measure is a very powerful tool. It solves a very old and very general problem in machine learning, namely the selection of a suitable type and level of abstraction, and can easily be applied to a wide range of inductive learning problems.

The experimental evaluation of our theorem prover, E/TSM, has shown that our approach to learning search control knowledge works very well. The prover can typically prove more than twice as many hard problems with the learning strategy than with the base strategy used to generate the training examples, despite the fact that the use of learned knowledge has both a significant startup cost and leads to a lower inference speed.

This success implies that the premises of our approach are correct. In particular, the representation of proof experiences as a set of annotated clauses and the transformation of clauses into representative patterns maintain a large amount of information about the value

of different search decisions. Term space mapping is able to transform this information into operative knowledge useful for the evaluation of new clauses.

In addition to the general relevance of the relative information gain, much of our other work can be applied to various related fields where forward-chaining search processes need to be controlled. This includes in particular other saturating theorem provers, e.g. resolution-based systems or inductive theorem provers, but also many rule-based expert systems and automated planning systems. It also can be extended to the bottom-up component of systems combining both forward and backward reasoning, as implemented in cooperative theorem provers.

Appendix A

The E Equational Theorem Prover – Conventional Features

E is a purely equational theorem prover, based on ordered paramodulation and rewriting. It is based on the superposition calculus **SP** described in Section 2.7 and the given-clause algorithm from Figure 4.2 on page 44. The unique features of the prover are the *perfectly shared term representation* and the optimizations enabled by this, and the very flexible and powerful interface for integrating and selecting search control heuristics.

The proof procedure is realized on a layer of libraries roughly depicted in Figure A.1. The infrastructure layer implements services and data types used by all other modules. Important among the services are an efficient memory management subsystem, stream abstractions for file input, and a generic scanner for lexical analysis of data from arbitrary sources. Important data types are splay trees (statistically balanced binary search trees [ST85]) for different key/value pairs, dynamic strings with reference counting, unlimited stacks, queues and dynamic arrays.

The next layer implements more specialized data types for theorem proving, like terms, equations, clauses and evaluation trees. It also implements basic operations, like term replacing, matching, unification and term orderings. Based on this layer, a separate module implements the basic inferences of the calculus. These two layers form the core inference engine discussed in the next section.

On the same level as the inference module is a module for heuristics and strategies, which is described in Section A.2. It implements various literal selection functions and the clause selection mechanism.

Finally, we have implemented the proof procedure on top of the other modules.

The theorem prover is implemented in about 70,000 lines of ISO/ANSI C, and is widely portable among current UNIX dialects. The code used exclusively for the learning component consists of about 12,000 lines of code and makes extensive use of the lower library layers.

Separate programs are used for the prover evaluation and for automatically generating the prover configuration scheme from test results (see Section A.2.3). They are implemented in the GNU dialect of AWK [AKW88].

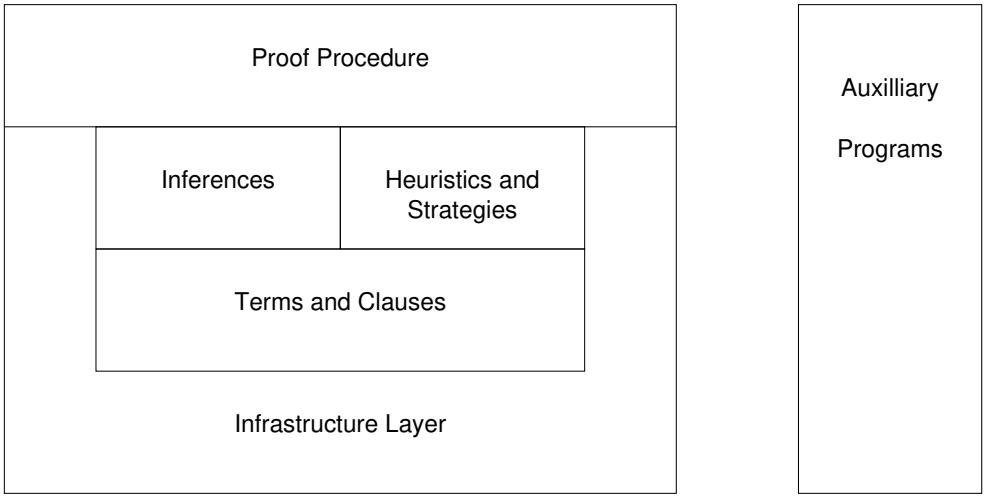


Figure A.1: Software architecture of E

A.1 Inference Engine

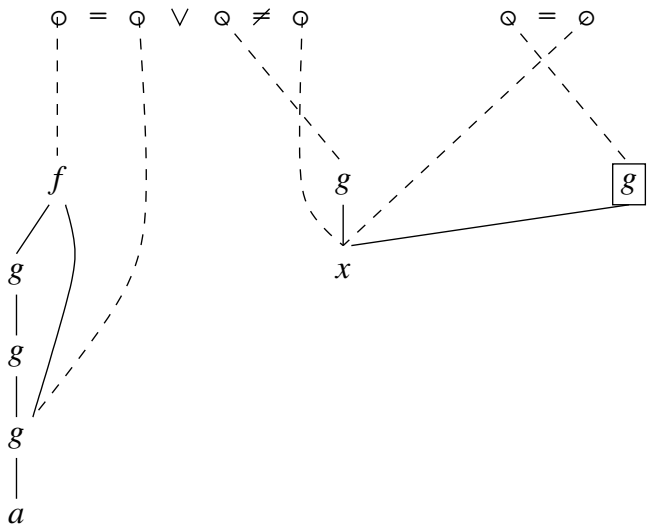
As described above, E is directly based on the **SP** calculus. Most inferences are implemented in a straightforward way. The prover does not implement any special inferences for non-equational literals. However, since any inference is typically followed by clause normalization and elimination of redundant literals, superposition and equality resolution simulate resolution inferences relatively closely.

A.1.1 Shared terms and rewriting

The inference engine of E is built around *perfectly shared terms* as the core data type. That means that any unique subterm in the current clause set is only represented once. Exceptions are only short-lived temporary clause copies for generating inference between different instances of the same clause, and individual term nodes that represent top positions of maximal terms in literals eligible for resolution and hence can be rewritten only under stricter conditions.

The shared term data structure is realized as a general *term bank* where terms are indexed by top symbol, a selectable set of flags to differentiate between otherwise identical terms (used only to differentiate between top terms of literals eligible for resolution in E), and pointers to the argument terms. A combination of hashing and splay trees is used for efficient access to the terms stored in the bank. Terms are administrated using reference counting and superterm-pointers. Consequently, they are inserted bottom-up and removed top-down.

Example: Consider the two clauses
$\underline{g(x)} \simeq x$ and



Remarks: Solid lines indicate term-subterm relationships, dashed lines indicate term-in-literal relationships. The boxed term node corresponds to a top position of a maximal term in a literal eligible for resolution.

Figure A.2: Shared term representation in E

$f(g(g(g(a))), g(a)) \simeq g(a) \vee g(x) \not\simeq a.$
Maximal terms in literals eligible for resolution are marked by underlining.

Figure A.2 shows the graph representation of the clauses, the following table shows a possible representation of the term set in a term bank:

Address	Top symbol	Flags	Subterms	Represented term
1	x	0	-	x
2	a	0	-	a
3	g	0	*1	$g(x)$
4	g	1	*1	$\underline{g(x)}$
5	g	0	*2	$\overline{g(a)}$
6	g	0	*5	$g(g(a))$
7	g	0	*6	$g(g(g(a)))$
8	f	0	*7,*5	$f(g(g(g(a))), g(a))$

Given this term bank, the two clauses are represented as $*4 \simeq *1$ and $*8 \simeq *5 \vee *3 \not\simeq *2$.

In normal proof searches, term sharing can reduce the number of term nodes needed to represent a search state between 10 and 1000 fold. It is quite typical that less than two term nodes are needed to represent the terms in a literal, i.e. the amount of memory taken up by term nodes grows at most linearly with the number of literals. In practice, memory

consumed by term representations is not a dominant factor. This differs strongly from our experience with e.g. DISCOUNT, where term representations are the single most critical data structure for memory consumption.

As term nodes are typically shared between a large number of clauses, we can afford to store several pre-computed values with each term. In our case this includes the term weight (which is computed automatically during normal from building), a flag to denote reducibility with respect to a currently investigated rule or equation, and, most importantly, *normal form dates* for different rewrite relations (see the next section).

E not only shares terms to save memory, but also performs rewriting on the shared term representation. If a rewrite rule is applied to any subterm in any clause, all shared occurrences of this subterm in all clauses will be replaced. As this may influence superterms, the change is propagated recursively to all superterms. This may even lead to the collapse of large parts of the term bank.

Example: Consider again the two clauses from the previous example. If we use the unit-clause as a rewrite rule to replace $g(a)$ by a in the second clause, $f(g(g(g(a))), g(a)) \simeq g(a) \vee g(x) \not\simeq a$, this clause immediately collapses to $f(a, a) \simeq \vee g(x) \not\simeq a$.

To replace $g(a)$ with a, we replace all pointers to $*5$ with pointers to $*2$ in the term bank and check the affected superterms for existing duplicates in the term bank. In this case, the entry for $g(g(a))$ at address 6 becomes identical to the originally replaced entry at address 5. Thus, we recursively replace $*6$ with $*2$ (as the replacement for $*5$). Now the same effect happens at address 7. After we resolve this in a similar way, we can remove all terms no longer referenced.

The modified term bank looks like this:

Address	Top symbol	Flags	Subterms	Represented term
1	x	0	-	x
2	a	0	-	a
3	g	0	*1	$g(x)$
4	g	1	*1	$\underline{g(x)}$
5				
6				
7				
8	f	0	*2,*2	$f(a, a)$

The term bank representation of the affected clause has changed to $*8 \simeq *2 \vee *3 \not\simeq *2$.

Note that rewriting $g(x)$ in this clause does not have any effect on the left hand side of the rewrite rule, as the two terms are *not* shared in this case.

Shared rewriting significantly speeds up normal form building. Normal forms need only be computed once for each term, and similarly the reducibility of a term with a new clause (used in backward-contraction, i.e. the removing of rewritable clauses from the set of processed clauses) has to be checked only once for each term node.

A.1.2 Matching and unification

Matching is at the core of most contracting inferences, unification at the core of most generating ones. Since generating inferences are only performed between the selected clauses, the effort for contraction usually outweighs the effort for generation by far. In particular, we found unification to be very cheap despite its theoretically exponential behaviour. Consequently, unification is implemented in a straightforward manner. Nevertheless, it is still less costly than e.g. the checking of ordering constraints or even the construction of new terms for newly generated clauses.

Matching attempts, on the other hand, are a major contributor to the overall CPU usage of most theorem provers which perform rewriting. While each individual match attempt is cheap, the search for matching rules and equations from the set of processed clauses is quite expensive. We have therefore implemented an indexing scheme that makes use of our shared term representation to optimize the access to these clauses.

The aim of an index for rewriting is the following: Given a term t and a set of unit-clauses P, find (sequentially or all at once) all clauses $l \simeq r$ from P such that $\sigma(l) = t$.

Following the extremely impressive results of Waldmeister, we have chosen a *perfect discrimination tree* (see [Gra95, GF98]) as the core data structure for our indexing algorithm. Perfect discrimination trees are a *perfect filter* for terms, i.e. they find only matching term, and can construct the match during the search. Moreover, they are easy to modify if new terms are inserted or old terms (or clauses) are deleted from the index.

A perfect discrimination tree basically treats a term as linear word, and branches on the symbol at each position in this word. Each branch in the tree thus represents a set of terms with a common initial sequence. What is new in E is that we maintain a monotonously increasing time counter for each proof search. This counter is increased whenever a new, non-trivial unit-clause is selected for processing, i.e. whenever the rewrite relation is about to change. We also store the normal form date of each term with respect to orientable unit clauses and with respect to all unit clauses with each term. Branches in the discrimination tree are annotated with the time at which the the youngest clause indexed by this branch was selected and the weight of the smallest indexed term.

This data can be used to cut off branches of the tree early. The following (somewhat artificial) example illustrates this point:

Example: Consider the following set of rewrite-rules, where the selection date of each clause is given by its running number and the weight given in brackets is computed by counting 2 for each function symbol and 1 for each variable symbol in the left hand side of the rule:

1. $f(x, b) \to e$ (5)
2. $f(x, c) \to e$ (5)
3. $f(x, d) \to e$ (5)
4. $f(a, g(a)) \to e$ (8)

5. $f(a, g(g(a))) \to e$ (10)
6. $f(a, g(g(g(a)))) \to e$ (12)
7. $f(y, y) \to e$ (4)

Figure A.3 shows the resulting *constrained perfect discrimination tree*.

If we want to find a match for $f(a, a)$ with normal form date 4 and weight 6, we can immediately eliminate the branch starting with $f - x$ due to the age constraint. Similarly, we can eliminate the branch starting with $f - a$ due to the size constraints. Thus, the only match, $f(y, y) \to e$ is found without any backtracking.

The pruning of older clauses is particularly effective in combination with the shared terms, as a large number of terms for which we want to find a potential rewrite rule have been brought into normal form at some earlier date.

Our indexing scheme is efficient enough to ensure the time for rewriting is usually dominated not by the search for matching clauses but by the ordering comparisons necessary for rewriting with unorientable unit clauses. In fact, for many problems with a large number of unit equality clauses it pays to use only orientable clauses for normalizing unprocessed new clauses for evaluation. This is facilitated in E by using two separate indexes for orientable and unorientable positive unit clauses.

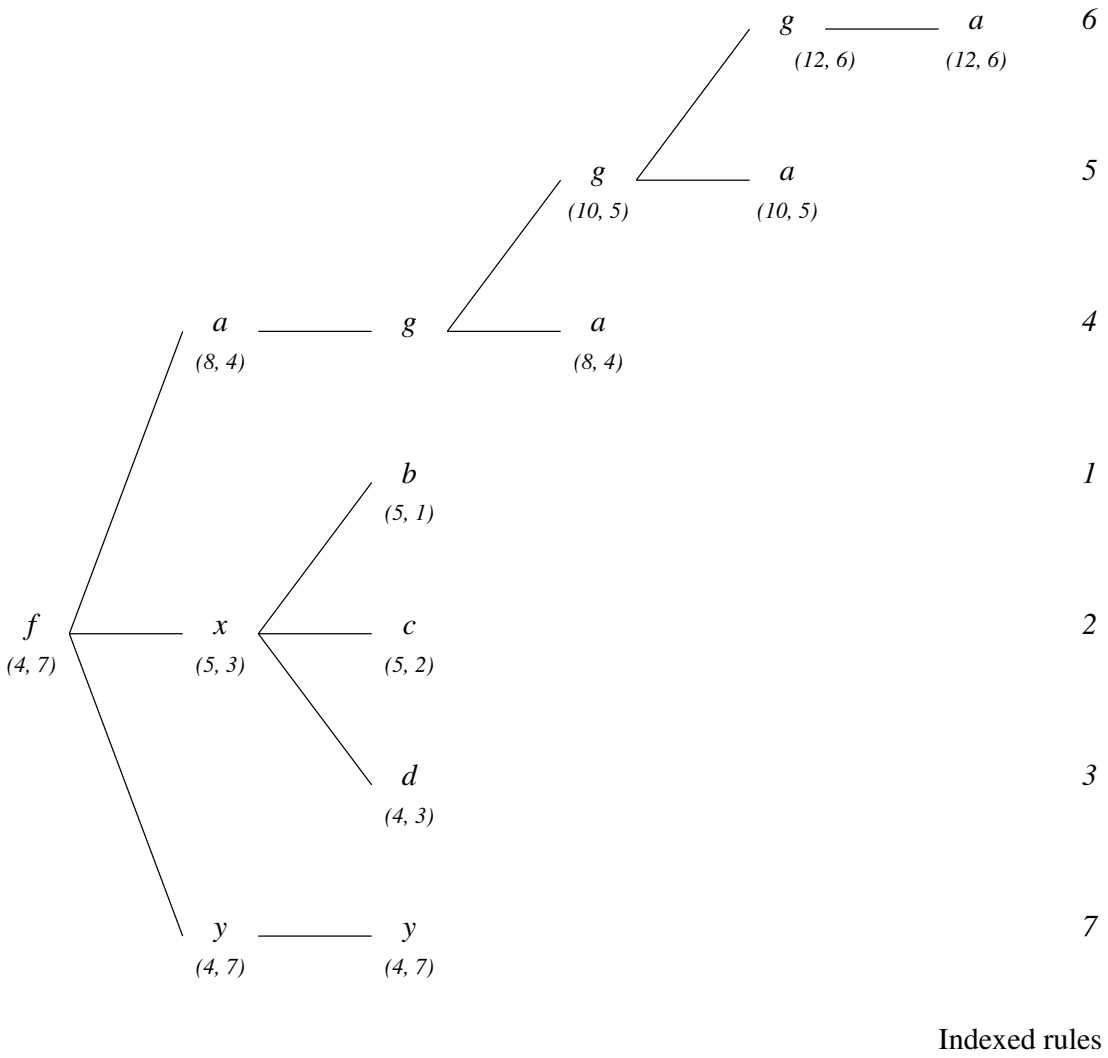
We use the same indexing structure for all contracting inferences with unit clauses: Rewriting, simplify-reflect, and equality subsumption. However, for simplify-reflect and equality subsumption we can use only weight constraints.

A.1.3 Term orderings

E supports two kinds of reduction orderings: The lexicographic term ordering (LPO), suggested by Kamin and Levi as a variant of the *Recursive Path Ordering* [Der79], and the Knuth-Bendix-Ordering (KBO) [KB70].

The LPO is parameterized by a precedence on the function symbols, the KBO by a precedence and a set of weights for the function symbol. E currently does not allow the user to explicitly set weights or precedence. It does contain a selection of simple algorithms that generate weight and precedence based on properties of the symbols and the problem specification, such as arity of the symbol or frequency of occurrence. The default term ordering used in all our experiments is a KBO. In the default precedence unary symbols are the largest, all other symbols are ordered by arity (i.e. a symbol with larger arity is larger than a symbol with smaller arity). Order between symbols with the same arity is decided by order of appearance. The largest non-constant symbol (which is usually unary) is assigned a weight of 0, all others a weight of 1. This ordering copes well with group-like structures with an inverse element, which occur quite frequently in real proof problems.

The current implementation of term orderings is straightforward, with only very limited optimizations. Consequently, ordering comparisons are one of the most costly operations



Remarks: Nodes are labeled with function symbols and tuples (*weight*,*date*), where *weight* is the weight of the smallest indexed term in the subtree and *date* is the time stamp of the youngest clause in the subtree.

Figure A.3: A constrained perfect discrimination tree

in E at the moment, and improvements in both the implementation (using caching to speed up comparisons) and the generation of good orderings are among our top priorities.

A.2 Search Control

A.2.1 Clause selection

E has a very flexible and powerful interface for specifying clause selection heuristics. It controls the selection of a clause using a weighted round-robin scheme with an arbitrary number of priority queues, where the order within each queue is determined by a priority function and and a weight function.

Priority functions

Priority functions assign one of a relatively small number of priorities to a clause. Some typical priority functions implemented in E are `ConstPrio`, `PreferGoals`, `PreferNonGoals` and `PreferUnitGroundGoals`.

`ConstPrio` assigns the same priority to all clauses. Combining this priority function with one or two weight functions simulates the search control of most existing theorem provers.

`PreferGoals` assigns a high priority to all negative clauses (potential goals) and a low priority to all other clauses. `PreferNonGoals` behaves in exactly the opposite way. These functions can e.g. be used to emulate the behaviour of DISCOUNT on problems with non-ground goals (compare Section 4.3).

Finally, `PreferUnitGroundGoals` will always prefer unit ground goals. It can be used to emulate the behaviour of a traditional completion-based prover for goals without variables, where all processed clauses are immediately used to rewrite the goal. However, it has also proven to be quite useful for the general case.

For a complete overview of available priority functions see [Sch99a].

Weight functions

Weight functions are the most important means of ordering clauses. They assign a numerical evaluation, i.e. a (real) number to a clause. For most weight functions, this weight is based on syntactic properties of the clause, however, some weight functions also consider the state of the proof search. The learning heuristics described in this thesis are used to implemented weight functions.

We will only describe the most important weight functions here – for a more complete overview again see [Sch99a]. There are three primary generic weight functions. These are `Clauseweight`, `Refinedweight` and `FIFOWeight`.

`Clauseweight` takes three arguments: A weight for function symbols w_f, a weight for variable symbols w_v, and a multiplier m_p applied to positive literals. It returns the sum of the term weights of the terms in negative literals (see Definition 4.6) plus the sum of the weights of the terms in positive literals times m_p as the weight of a clause. Thus, it realizes a slightly generalized version of simple symbol counting.

`Refinedweight` is a very similar weight function. It differs in that it uses two additional arguments, m_t and m_l. These are applied to maximal terms and maximal literals (in the used term ordering or its extension to literals), respectively.

`FIFOWeight` finally is a very simple function that just returns an monotonically increasing value for each new clause it evaluates. Thus, it realizes the first-in first-out search heuristic.

Composite search heuristics

As we stated above, complete search heuristics are defined by a set of priority queues and a weighted round-robin scheme. Each queue is ordered according to an *evaluation function*, which combines a priority function and a weight function. A general specification of a search heuristic consists of a weighted list of evaluation functions.

Example: The `Default` search heuristic used if neither automatic mode or a specific heuristic are chosen by the user is specified as

```
(3*Refinedweight(PreferNonGoals,2,1,1.5,1.1,1),
1*Refinedweight(PreferGoals,1,1,1.5,1.1,1.1)).
```

If this heuristic is chosen, 3 out of 4 clauses are chosen from the non-negative clauses (unless none are present), the last clause is selected from the set of negative clauses. Clauses selected according to the first evaluation functions are evaluated with the `Refinedweight` weight function with $w_f = 2$, $w_v = 1$, $m_t = 1.5$, $m_l = 1.1$, and $m_p = 1$. Other clauses are evaluated similarly with $w_f = 1$, $w_v = 1$, $m_t = 1.5$, $m_l = 1.1$, and $m_p = 1.1$.

E has some other predefined heuristics, two of which are referenced throughout this thesis. The first one, `Weight`, is equivalent to `(1*Clauseweight(PreferUnitGroundGoals, 2,1,1))`[1], which closely emulates the behaviour of the completion-based prover DISCOUNT in default mode. The second frequently used search heuristic is `Standardweight`, defined as `(1*Clauseweight(ConstPrio,2,1,1))`. `Standardweight` closely models the search heuristic of most traditional saturating theorem provers based solely on symbol counting.

A.2.2 Literal selection

The standard superposition calculus (described e.g. in [BG94]) allows the selection of arbitrary negative literals. For **SP** we have extended this and, under certain circumstances, allow the additional selection of *positive* literals. E makes use of this double freedom and implements a large number of different literal selection strategies.

[1] In this case, $w_f = 2$, $w_v = 1$, and $m_p = 1$.

We will only describe some of these strategies that are of particular interest. A complete overview is again available in [Sch99a].

First, the `NoSelection` strategy will not use literal selection at all, but rather implements the standard superposition calculus without selection.

The `SelectNegativeLiterals` selection scheme will always select all negative literals. In this way, it implements a maximum literal positive unit strategy [Der91] in the Horn case. It implements the least restricted positive strategy in the general case.

The `SelectLargestNegLit` strategy selects the largest literal (by weight of the terms) if at least one negative literal occurs in the clause. In case of ties it picks an arbitrary literal among the largest ones. This is very simple, but quite successful general purpose literal selection strategy.

Finally, `SelectNonRROptimalLit` is a more complex literal selection scheme that dynamically decides if a literal shall be selected at all. If the clause is *range-restricted* (i.e. if all variables occurring in negative literals also occur in positive literals), then no selection takes place. If the clause contains negative literals and is not range-restricted, then if there are negative ground literals, the negative ground literal with the largest weight difference between both sides is selected. If there are negative literals, but no negative ground literals, the negative literal with the largest weight difference between both sides is selected.

The rationale for this scheme is easy to explain: First, range-restricted Horn-clauses can be seen as *procedures*, where the head (the positive literal) does the variable binding and the tail (the remaining negative literals) realize the body of the procedure. If we view the a clause in this way, not using the head literal for paramodulating into another clause is counterproductive. Note that for non-Horn clauses, this argument does not hold (a stronger form of restriction might be useful here), however, we treat them in the same way for consistencies sake.

If we select negative literals in all clauses that have negative literals, a clause can only be used for paramodulation into another clause, if it has no negative literals. In other words, if we want to use a clause, we need to solve *all* of its negative literals. Ground literals can either be solved without further instantiation, or cannot be solved at all. Therefore there is usually much less effort associated with solving a ground literal than with solving a non-ground literal. We therefore select ground literals first. Moreover, if we need to solve all literals, we in particular need to solve the most difficult literal. It therefore makes sense to delay all work on the clause until this literal has been solved. In equational theorem proving, solving a negative literal means we need to show that both sides of the literal are equal. We use the size difference as a very simple approximation for the difficulty of this task. This explains why we select literals with a large weight difference between the two sides first.

A.2.3 Automatic prover configuration

There is a very wide range of proof problems, from a variety of domains and with very different syntactic and semantic properties. No single proof search strategy or heuristic can give optimal performance in all different cases. Therefore most leading theorem provers

feature an *automatic mode* in which the prover analyzes certain problem characteristics and selects a (hopefully) suitable strategy. We have implemented a similar mode for E. It partitions the space of all problems according to 8 criteria:

1. The most important criterion is the type of the axioms, i.e. the least specific type of any non-negative clause in the original specification. Possible values are *unit*, *Horn*, and *general*.

2. Similarly, the type of the goal or goals (all negative clauses) is categorized into one of the values *unit* and *Horn*.

3. The third feature is the equality content. We distinguish between problems *without equality*, with *some equality* literals, and *pure equality* problems.

4. The next feature evaluates the number of positive non-ground unit clauses. We found that this feature is very helpful for choosing a literal selection function. Possible values are *few*, *some*, and *many* such clauses, the exact limits for each category depend on the axioms' types.

5. The fifth criterion again is determined by the goals. It distinguishes between *ground* goals and *non-ground* goals. This feature is particularly important for unit-equality problems, where ground goals can be proveed with pure unfailing completion, while non-ground goals need a more general strategy.

6. Another relevant feature is the *number of clauses* in the initial specification. Possible values of this feature are again *few* (less than 30), *some* (more than 30 but less than 150) and *many* clauses.

7. Similarly, we count the number of literals in the initial clauses. Limits here are 15 for *few* and *100* to discriminate between *some* and *many* literals.

8. The last feature is determined by the number of term cells in the initial axioms. We consider problems with less than 60 term cells to have *small* terms, problems with more than 60 but less than 1000 term cells to have *medium* terms, and problems with more than 1000 term cells to have *large* terms.

We have implemented a program that automatically determines good values for the clause selection heuristic, the literal selection strategy and some secondary parameters for each of the classes spanned by this parameter space from the results of standardized test runs. While the 8 features partition the potential space of all problems into 2916 subclasses, only about 150 of them contain any TPTP problems. E 0.51 needs only 34 distinct strategies to cover these classes, including default strategies for the empty classes.

Appendix B

Specification of Proof Problems

In this appendix we give a short description and (except for the SET103-6 problems) the complete specification for the proof problems frequently referenced in the main text. All problems except for INVCOM and LUSK6 are taken from the TPTP problem library, version 2.1.0 [SS97b].

B.1 INVCOM

INVCOM is a very simple problem in the domain of groups. The equational specification of a group goes back to [KB70]. We have added a very simple hypothesis: Multiplication of an element with the corresponding inverse element is commutative. The resulting problem can be solved by any state-of-the-art prover with support for equality in trivial time and with a very short search derivation.

```
#----------------------------------------------------------------
# Equational specification of a group with simple hypothesis
#
# Only unit clauses in infix-equational notation.
#

# There exists a right-neutral element (0).
f(X,0)=X.

# For each X, there is a right inverse element i(X).
f(X,i(X))=0.

# f is associative.
f(f(X,Y),Z)=f(X,f(Y,Z)).

# Skolemized and inverted hypothesis: Multiplication with inverse
# element is commutative.

f(a,i(a)) != f(i(a),a).
```

B.2 BOO007-2

BOO007-2 is a unit-equality problem of medium difficulty taken from the TPTP version 2.1.0. The aim is to show that the multiplicative operator in a boolean algebra is associative.

```
#------------------------------------------------------------------------
# File     : BOO007-2 : TPTP v2.1.0. Released v1.0.0.
# Domain   : Boolean Algebra
# Problem  : Product is associative ( (X * Y) * Z = X * (Y * Z) )
# Version  : [ANL] (equality) axioms.
# English  :
#
# Refs     : [Ver92] Veroff (1992), Email to G. Sutcliffe
# Source   : [Ver92]
# Names    : associativity [Ver92]
#
# Status   : unsatisfiable
# Rating   : 0.33 v2.1.0, 0.75 v2.0.0
# Syntax   : Number of clauses    :   15 (   0 non-Horn;  15 unit;
#                                         1 RR)
#            Number of literals   :   15 (  15 equality)
#            Maximal clause size  :    1 (   1 average)
#            Number of predicates :    1 (   0 propositional;
#                                         2-2 arity)
#            Number of functors   :    8 (   5 constant; 0-2 arity)
#            Number of variables  :   24 (   0 singleton)
#            Maximal term depth   :    3 (   2 average)
#
# Comments :
#          : tptp2X -f setheo:sign -t rm_equality:rstfp BOO007-2.p
#------------------------------------------------------------------------

# commutativity_of_add, axiom.
equal(add(X, Y), add(Y, X)) <- .

# commutativity_of_multiply, axiom.
equal(multiply(X, Y), multiply(Y, X)) <- .

# distributivity1, axiom.
equal(add(multiply(X, Y), Z), multiply(add(X, Z), add(Y, Z))) <- .

# distributivity2, axiom.
equal(add(X, multiply(Y, Z)), multiply(add(X, Y), add(X, Z))) <- .

# distributivity3, axiom.
equal(multiply(add(X, Y), Z), add(multiply(X, Z), multiply(Y, Z))) <- .

# distributivity4, axiom.
equal(multiply(X, add(Y, Z)), add(multiply(X, Y), multiply(X, Z))) <- .
```

```
# additive_inverse1, axiom.
equal(add(X, inverse(X)), multiplicative_identity) <- .

# additive_inverse2, axiom.
equal(add(inverse(X), X), multiplicative_identity) <- .

# multiplicative_inverse1, axiom.
equal(multiply(X, inverse(X)), additive_identity) <- .

# multiplicative_inverse2, axiom.
equal(multiply(inverse(X), X), additive_identity) <- .

# multiplicative_id1, axiom.
equal(multiply(X, multiplicative_identity), X) <- .

# multiplicative_id2, axiom.
equal(multiply(multiplicative_identity, X), X) <- .

# additive_id1, axiom.
equal(add(X, additive_identity), X) <- .

# additive_id2, axiom.
equal(add(additive_identity, X), X) <- .

# prove_associativity, conjecture.
<- equal(multiply(a, multiply(b, c)), multiply(multiply(a, b), c)).

#-----------------------------------------------------------------
```

B.3 LUSK6

The LUSK6 example is one of the problems presented in [LO82], an equivalent version is contained in TPTP 2.1.0 as RNG009-5. The hypothesis is that in a ring with $x^3 = x$ for all x the multiplicative operator is commutative. This is a challenging problem even for most specialized unit-equational provers. The first known automatic proof for this problem was found by DISCOUNT in 1995. DISCOUNT needs about 400 seconds to prove this problem on a current SUN SPARCStation Ultra-10/300. E and Waldmeister can prove the the problem in about 10 seconds on identical hardware.

```
#-----------------------------------------------------------------
#   In a ring, if x*x*x = x for all x in the ring, then
#   x*y = y* for all x,y in the ring.
#

j(0,X)      = X.                # 0 is a left identity for sum
j(X,0)      = X.                # 0 is a right identity for sum
j(g(X),X)   = 0.                # There exists a left inverse for sum
j(X,g(X))   = 0.                # There exists a right inverse for sum
```

```
j(j(X,Y),Z) = j(X,j(Y,Z)).      # Associativity of addition
j(X,Y)      = j(Y,X).           # Commutativity of addition
f(f(X,Y),Z) = f(X,f(Y,Z)).      # Associativity of multiplication
f(X,j(Y,Z)) = j(f(X,Y),f(X,Z)). # Distributivity axioms
f(j(X,Y),Z) = j(f(X,Z),f(Y,Z)). #
f(f(X,X),X) = X.                # Special hypothesis: x*x*x = x

f(a,b) != f(b,a).               # Hypothesis
```

B.4 HEN011-3

The problem HEN011-3 is a Horn problem with equality. Despite its relatively small axiomatization, it is fairly hard. The problem specifies a certain division operation and shows that this is symmetric. Details can be found in [MOW76].

```
#----------------------------------------------------------------------------
# File     : HEN011-3 : TPTP v2.1.0. Released v1.0.0.
# Domain   : Henkin Models
# Problem  : This operation is commutative
# Version  : [MOW76] axioms.
# English  : Define & on the set of Z', where Z' = identity/Z,
#            by X' & Y'=X'/(identity/Y'). The operation is commutative.

# Refs     : [MOW76] McCharen et al. (1976), Problems and Experiments for a
# Source   : [ANL]
# Names    : HP11 [ANL]

# Status   : unsatisfiable
# Rating   : 0.83 v2.1.0, 1.00 v2.0.0
# Syntax   : Number of clauses    :   13 (   0 non-Horn;  10 unit;   9 RR)
#            Number of literals   :   17 (   9 equality)
#            Maximal clause size  :    3 (   1 average)
#            Number of predicates :    2 (   0 propositional; 2-2 arity)
#            Number of functors   :    9 (   8 constant; 0-2 arity)
#            Number of variables  :   13 (   3 singleton)
#            Maximal term depth   :    4 (   1 average)

# Comments :
#          : tptp2X -f setheo:sign -t rm_equality:rstfp HEN011-3.p
#----------------------------------------------------------------------------
# Quotient_less_equal1, axiom.
equal(divide(X, Y), zero) <-
    less_equal(X, Y).

# quotient_less_equal2, axiom.
less_equal(X, Y) <-
    equal(divide(X, Y), zero).

# quotient_smaller_than_numerator, axiom.
```

```
less_equal(divide(X, Y), X) <- .

# quotient_property, axiom.
less_equal(divide(divide(X, Z), divide(Y, Z)), divide(divide(X, Y), Z)) <- .

# zero_is_smallest, axiom.
less_equal(zero, X) <- .

# less_equal_and_equal, axiom.
equal(X, Y) <-
    less_equal(X, Y),
    less_equal(Y, X).

# identity_is_largest, axiom.
less_equal(X, identity) <- .

# part_of_theorem, hypothesis.
 <- equal(divide(divide(identity, a), divide(identity,
    divide(identity, b))), divide(divide(identity, b),
    divide(identity, divide(identity, a)))).

# identity_divide_a, hypothesis.
equal(divide(identity, a), c) <- .

# identity_divide_b, hypothesis.
equal(divide(identity, b), d) <- .

# identity_divide_c, hypothesis.
equal(divide(identity, c), e) <- .

# identity_divide_d, hypothesis.
equal(divide(identity, d), g) <- .

# prove_commutativity, conjecture.
 <- equal(divide(c, g), divide(d, e)).

#----------------------------------------------------------------------------
```

B.5 PUZ031-1

PUZ031-1 is a classical problem for testing the performance of automated theorem provers, also known as Schubert's Steamroller. It is the formalization of a logical puzzle and encodes a set of food-chain relationships between animals and grains. The problem contains unit, Horn and non-Horn clauses, but no equality. A more detailed description can be found in [Sti86]. Most current theorem provers can prove this problem in a few seconds.

```
#----------------------------------------------------------------------------
# File     : PUZ031-1 : TPTP v2.1.0. Released v1.0.0.
# Domain   : Puzzles
```

```
# Problem  : Schubert's Steamroller
# Version  : Special.
# English  : Wolves, foxes, birds, caterpillars, and snails are animals, and
#            there are some of each of them. Also there are some grains, and
#            grains are plants. Every animal either likes to eat all plants
#            or all animals much smaller than itself that like to eat some
#            plants. Caterpillars and snails are much smaller than birds,
#            which are much smaller than foxes, which in turn are much
#            smaller than wolves. Wolves do not like to eat foxes or grains,
#            while birds like to eat caterpillars but not snails.
#            Caterpillars and snails like to eat some plants. Therefore
#            there is an animal that likes to eat a grain eating animal.

# Refs     : [Sti86] Stickel (1986), Schubert's Steamroller Problem: Formul
#          : [Pel86] Pelletier (1986), Seventy-five Problems for Testing Au
#          : [WB87]  Wang & Bledsoe (1987), Hierarchical Deduction
#          : [MB88]  Manthey & Bry (1988), SATCHMO: A Theorem Prover Implem
# Source   : [Pel86]
# Names    : Pelletier 47 [Pel86]
#          : steamroller.ver1.in [ANL]
#          : steam.in [OTTER]
#          : SST [WB87]

# Status   : unsatisfiable
# Rating   : 0.22 v2.1.0, 0.00 v2.0.0
# Syntax   : Number of clauses    :   26 (   1 non-Horn;   6 unit;  26 RR)
#            Number of literals   :   63 (   0 equality)
#            Maximal clause size  :    8 (   2 average)
#            Number of predicates :   10 (   0 propositional; 1-2 arity)
#            Number of functors   :    8 (   6 constant; 0-1 arity)
#            Number of variables  :   33 (   0 singleton)
#            Maximal term depth   :    2 (   1 average)

# Comments : This problem is named after Len Schubert.
#          : tptp2X -f setheo:sign -t rm_equality:rstfp PUZ031-1.p
#--------------------------------------------------------------------------
# wolf_is_an_animal, axiom.
animal(X) <-
    wolf(X).

# fox_is_an_animal, axiom.
animal(X) <-
    fox(X).

# bird_is_an_animal, axiom.
animal(X) <-
    bird(X).

# caterpillar_is_an_animal, axiom.
animal(X) <-
    caterpillar(X).
```

```
# snail_is_an_animal, axiom.
animal(X) <-
    snail(X).

# there_is_a_wolf, axiom.
wolf(a_wolf) <- .

# there_is_a_fox, axiom.
fox(a_fox) <- .

# there_is_a_bird, axiom.
bird(a_bird) <- .

# there_is_a_caterpillar, axiom.
caterpillar(a_caterpillar) <- .

# there_is_a_snail, axiom.
snail(a_snail) <- .

# there_is_a_grain, axiom.
grain(a_grain) <- .

# grain_is_a_plant, axiom.
plant(X) <-
    grain(X).

# eating_habits, axiom.
eats(Animal, Plant);
eats(Animal, Small_animal) <-
    animal(Animal),
    plant(Plant),
    animal(Small_animal),
    plant(Other_plant),
    much_smaller(Small_animal, Animal),
    eats(Small_animal, Other_plant).

# caterpillar_smaller_than_bird, axiom.
much_smaller(Catapillar, Bird) <-
    caterpillar(Catapillar),
    bird(Bird).

# snail_smaller_than_bird, axiom.
much_smaller(Snail, Bird) <-
    snail(Snail),
    bird(Bird).

# bird_smaller_than_fox, axiom.
much_smaller(Bird, Fox) <-
    bird(Bird),
    fox(Fox).
```

```
# fox_smaller_than_wolf, axiom.
much_smaller(Fox, Wolf) <-
    fox(Fox),
    wolf(Wolf).

# wolf_dont_eat_fox, axiom.
 <- wolf(Wolf),
    fox(Fox),
    eats(Wolf, Fox).

# wolf_dont_eat_grain, axiom.
 <- wolf(Wolf),
    grain(Grain),
    eats(Wolf, Grain).

# bird_eats_caterpillar, axiom.
eats(Bird, Catapillar) <-
    bird(Bird),
    caterpillar(Catapillar).

# bird_dont_eat_snail, axiom.
 <- bird(Bird),
    snail(Snail),
    eats(Bird, Snail).

# caterpillar_food_is_a_plant, axiom.
plant(caterpillar_food_of(Catapillar)) <-
    caterpillar(Catapillar).

# caterpillar_eats_caterpillar_food, axiom.
eats(Catapillar, caterpillar_food_of(Catapillar)) <-
    caterpillar(Catapillar).

# snail_food_is_a_plant, axiom.
plant(snail_food_of(Snail)) <-
    snail(Snail).

# snail_eats_snail_food, axiom.
eats(Snail, snail_food_of(Snail)) <-
    snail(Snail).

# prove_the_animal_exists, conjecture.
 <- animal(Animal),
    animal(Grain_eater),
    grain(Grain),
    eats(Animal, Grain_eater),
    eats(Grain_eater, Grain).

#------------------------------------------------------------------------
```

B.6 SET103-6

The last problem, SET103-6, is selected from a set of very hard problems. They are based on a common specification of set theory described in [Qua92]. This specification contains 91 clauses with up to 5 literals, and a very large number of predicate and function symbols. Moreover, the specification includes fairly large equational literals. These properties lead to a very early explosion of the search space, and only fairly short proofs can be found for this class of problems. There is a total of 886 problems in TPTP 2.1.0 that use the same specification of set theory (with different queries and additional assumptions), i.e. these problems make up more than a quarter of the 3275 clause normal form problems in this TPTP release. Only 16.4% or 145 of these problems can be proved by E with the `StandardWeight` heuristic in 300 seconds, as opposed to 42.2% of the set of all clause normal form problems in this TPTP release.

SET103-6 is among those problems that can be solved by E with the standard heuristic. However, it still is a fairly hard problem for most state-of-the-art theorem provers.

As the full specification of the problem would take more than 8 pages, we refrain from including it. It is available with recent releases of the TPTP or on the WWW at
`http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/TPTP/SET/SET103-6+rm_eq_rstfp.lop`.

Bibliography

[AD93] J. Avenhaus and J. Denzinger. Distributing Equational Theorem Proving. In C. Kirchner, editor, *Proc. of the 5th RTA, Montreal*, number 690 in LNCS, pages 62–76. Springer, 1993. Also available as Seki-Report SR-93-06, Kaiserslautern, 1993.

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In J. Hsiang, editor, *Proc. of the 6th RTA, Kaiserslautern*, number 914 in LNCS, pages 397–402. Springer, 1995.

[AKW88] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[Ave95] J. Avenhaus. *Reduktionssysteme*. Springer, 1995.

[Bac98] L. Bachmair. Personal communication at CADE-15, Lindau. Unpublished, 1998.

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. Siekman, and V. Sorge. Ωmega: Towards a Mathematical Assistant. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 252–255. Springer, 1997.

[BDP89] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In *Coll. on the Resolution of Equations in Algebraic Structures, Austin, 1987*. Academic Press, 1989.

[BG90] L. Bachmair and H. Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, number 449 in LNAI, pages 427—441. Springer, 1990.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[BH96] A. Buch and Th. Hillenbrand. Waldmeister: Development of a high performance completion-based theorem prover. SEKI-Report SR-96-01, Fachbereich Informatik, Universität Kaiserslautern, 1996. Available at `http://agent.informatik.uni-kl.de/waldmeister/`.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 1998.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Boy92] T. Boy de la Tour. An Optimality Result for Clause Form Translation. *Journal of Symbolic Computation*, 14:283–301, 1992.

[Bra98] F. Brandt. Example Selection for Learning in Automated Theorem Proving. Diplomarbeit in Informatik, Institut für Informatik, TU München, 1998. (available from the author at `brandtf@informatik.tu-muenchen.de`).

[BRLP98] J. Van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining Synthesized Software. In *Proc. of the 13th IEEE Conference on Automated Software Engineering, Honolulu*. IEEE, 1998.

[Car86] J.G. Carbonell. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning*, volume 2. Morgan Kaufmann, 1986.

[CL73] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics. Academic Press, 1973.

[CN89] P. Clark and T. Niblett. The CN2 Induction Algorithm. *Machine Learning*, 3, 1989.

[CV88] J.G. Carbonell and M. Veloso. Integrating Derivational Analogy into a General Problem Solving Architecture. In *Proc. of the 1988 DARPA Workshop on Case-Based Reasoning, Clearwater Beach*, 1988.

[Den93] J. Denzinger. *Teamwork: Eine Methode zum Entwurf verteilter, wissensbasierter Theorembeweiser*. PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, 1993. (German Language).

[Den95] J. Denzinger. Knowledge-Based Distributed Search using Teamwork. In *Proc. of the ICMAS-95, San Francisco*, pages 81–88. AAAI Press, 1995.

[Der79] N. Dershowitz. Orderings for Term Rewriting Systems. In S.R. Kosaraju, editor, *Proc. of the 20th Anual Symposium on Foundations of Computer Science, San Juan*, 1979.

[Der91] N. Dershowitz. Ordering-Based Strategies for Horn Clauses. In J. Mylopoulos, editor, *Proc. of the 12th IJCAI, Sydney*, volume 1, pages 118–124. Morgan Kaufmann, 1991.

[DF94] J. Denzinger and M. Fuchs. Goal Oriented Equational Theorem Proving Using Team Work. In *Proc. of the KI94, Saarbrücken*, number 861 in LNAI, pages 343–354. Springer, 1994.

[DF98] J. Denzinger and M. Fuchs. A Comparison of Equality Reasoning Heuristics. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (2) of *Applied Logic Series*, chapter 5, pages 361–382. Kluwer Academic Publishers, 1998.

[DF99] J. Denzinger and D. Fuchs. Cooperation of Heterogeneous Provers. In *Proc. of the 16th IJCAI, Stockholm*. Morgan Kaufmann, 1999. (to appear).

[DFF97] J. Denzinger, Marc Fuchs, and M. Fuchs. High Performance ATP Systems by Combining Several AI Methods. In *Proc. of the 15th IJCAI, Nagoya*. Morgan Kaufmann, 1997.

[DFGS99] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999. (also to be published as a SEKI report).

[DGHW97] B.I. Dahn, J. Gehne, T. Honigmann, and A. Wolf. Integration of Automated and Interactive Theorem Proving in ILF. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 57–60. Springer, 1997.

[DK96] J. Denzinger and M. Kronenburg. Planning for Distributed Theorem Proving: The Teamwork Approach. In G. Görz and S. Hölldobler, editors, *Proc. of the 20th German Annual Conference on Artificial Intelligence, Dresden (KI-96)*, number 1137 in LNAI, pages 43–56. Springer, 1996.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[DM79] N. Dershowitz and Z. Manna. Proving Termination with Multiset-Orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DS94a] J. Denzinger and S. Schulz. Analysis and Representation of Equational Proofs Generated by a Distributed Completion Based Proof System. Seki-Report SR-94-05, Universität Kaiserslautern, 1994.

[DS94b] J. Denzinger and S. Schulz. Recording, Analyzing and Presenting Distributed Deduction Processes. In H. Hong, editor, *Proc. 1st PASCO, Hagenberg/Linz*, volume 5 of *Lecture Notes Series in Computing*, pages 114–123, Singapore, 1994. World Scientific Publishing.

[DS96a] J. Denzinger and S. Schulz. Learning Domain Knowledge to Improve Theorem Proving. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 62–76. Springer, 1996.

[DS96b] J. Denzinger and S. Schulz. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation*, 21(4/5):523–541, 1996.

[DS98] J. Denzinger and S. Schulz. Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts. *Journal of Information and Computation*, 1998. (accepted for publication).

[DW94] B.I. Dahn and A. Wolf. A Calculus Supporting Structured Proofs. *Journal of Information Processing and Cybernetics (EIK)*, 30(5–6):261–276, 1994. Akademie Verlag Berlin.

[DW97] B.I. Dahn and C. Wernhard. First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. In *Proceedings of the 1st FTP, Linz*, pages 58–62. RISC Linz, Austria, 1997.

[Etz93] O. Etzioni. A Structural Theory of Explanation-Based Learning. *Artificial Intelligence*, 60(1):93–139, 1993. Elsevier Science Publishers, Amsterdam, North-Holland.

[FF97] Marc Fuchs and M. Fuchs. Case-Based Reasoning for Automated Deduction. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 6–10. Florida AI Research Society, 1997.

[FF98] Marc Fuchs and M. Fuchs. Feature-Based Learning of Search-Guiding Heuristics for Theorem Provers. *AI Communications*, 11(3,4):175–189, 1998.

[FS97] B. Fischer and J. Schumann. SETHEO Goes Software-Engineering: Application of ATP to Software Reuse. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 65–68. Springer, 1997.

[Fuc95a] M. Fuchs. Learning Proof Heuristics by Adapting Parameters. In *Proc. of the 12th International Workshop on Machine Learning*, pages 235–243, San Mateo, CA, 1995. Morgan Kaufmann.

[Fuc95b] M. Fuchs. Learning Proof Heuristics by Adapting Parameters. In A. Prieditis and S. Russel, editors, *Proc. of the 12th International Conference on Machine Learning*, pages 235–243. Morgan Kaufmann, 1995.

[Fuc96] M. Fuchs. Experiments in the Heuristic Use of Past Proof Experience. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 523–537. Springer, 1996.

[Fuc97a] M. Fuchs. Automatic Selection of Search-Guiding Heuristics. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 1–5. Florida AI Research Society, 1997.

[Fuc97b] M. Fuchs. *Learning Search Heuristics for Automated Deduction.* Number 34 in Forschungsergebnisse zur Informatik. Verlag Dr. Kovač, 1997. Accepted as a Ph.D. Thesis at the Fachbereich Informatik, Universität Kaiserslautern.

[GF98] P. Graf and D. Fehrer. Term Indexing. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (2) of *Applied Logic Series*, chapter 5, pages 125–147. Kluwer Academic Publishers, 1998.

[GK96] C. Goller and A. Küchler. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In *Proc. of the ICNN-96*, volume 1, pages 347–352. IEEE, 1996.

[Gol91] C. Goller. Domänenspezifische Heuristiken zur Suchraumreduktion im Automatischen Theoremberweisen mit SETHEO. Diplomarbeit in Informatik, Institut für Informatik, TU München, 1991. (German language).

[Gol94] C. Goller. A Connectionist Control Component for the Theorem Prover SETHEO. In *Proc. of the ECAI'94 Workshop W14: Combining Symbolic and Connectionist Processing*, pages 99–93. ECAI in cooperation with AAAI and IJCAI, 1994.

[Gol99a] C. Goller. *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems.* Number 206 in DISKI. Infix, Sankt Augustin, 1999. Accepted as a Ph.D. Thesis at the Fakultät für Informatik, Technische Universität München.

[Gol99b] C. Goller. Learning Search-Control Heuristics for Automated Deduction Systems with Folding Architecture Networks. In *Proc. of the European Symposium on Artificial Neural Networs (ESANN 99)*, 1999.

[Gra95] P. Graf. *Term Indexing.* Number 1053 in LNAI. Springer, 1995.

[Ham96] B. Hammer. Universal Approximation of Mappings on Structured Objects using the Folding Architecture. Osnabrücker Schriften zur Mathematik Reihe P, Heft 183, Fachbereich Mathematik/Informatik, Universität Osnabrück, 49069 Osnabrück, 1996.

[HBF96] T. Hillenbrand, A. Buch, and R. Fettig. On Gaining Efficiency in Completion-Based Theorem Proving. In H. Ganzinger, editor, *Proc. of the 7th RTA, New Brunswick*, number 1103 in LNCS, pages 432–435. Springer, 1996.

[HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. System Abstract: Waldmeister – Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 232–236. Springer, 1999.

[HR87] J. Hsiang and M. Rusinowitch. On Word Problems in Equational Theories. In *Proc. of the 14th ICALP, Karlsruhe*, number 267 in LNCS, pages 54–71. Springer, 1987.

[HS97] B. Hammer and V. Sperschneider. Neural Networks can approximate Mappings on Structured Objects. In P.P.Wang, editor, *Proc. of the 2nd International Conference on Computational Intelligence and Neuroscience*, volume 2, pages 211–214, 1997.

[JHA+99] S. P. Jones, John Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. The Haskell 98 Report. Technical report, Haskell.org, 1999. Available from `http://www.haskell.org/`.

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Algebra*, pages 263–297. Pergamon Press, 1970.

[KG96] A. Küchler and C. Goller. Inductive Learning in Symbolic Domains Using Structure-Driven Recurrent Neural Networks. In G. Görz and S. Hölldobler, editors, *Proc. of the 20th German Annual Conference on Artificial Intelligence, Dresden (KI-96)*, number 1137 in LNAI, pages 183–198. Springer, 1996.

[Kol92] J.L. Kolodner. An Introduction to Case-Based Reasoning. *Artificial Intelligence Review*, 6:3–34, 1992.

[KW94] T. Kolbe and C. Walther. Reusing Proofs. In *Proc. of the 11th ECAI, Amsterdam*, pages 80–84. John Wiley & Sons, Ltd., 1994.

[KW96] T. Kolbe and C. Walther. Termination of Theorem Proving by Reuse. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 106–120. Springer, 1996.

[LO82] E.L. Lusk and R.A. Overbeek. A Short Problem Set for Testing Systems that Include Equality Reasoning. Technical report, Argonne National Laboratory, Illinois, 1982.

[Löc99] B. Löchner. Personal communication at the meeting of the *GI-Fachgruppe 1.2.1 Deduktionssysteme*, Kaiserslautern. Unpublished, 1999.

[Lov68] D.W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15(2), 1968.

[Lov78] D.W. Loveland. *Automated Theorem Proving: A Logical Basis*. North Holland, Amsterdam, 1978.

[LPP+94] M. Lowry, A. Philpot, T. Pressburger, I. Underwood, R. Waldinger, and M. Stickel. Amphion: Automatic Programming for the NAIF Toolkit. *NASA Science Information Systems Newsletter*, 31:22–25, 1994.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 1(8):183–212, 1992.

[McC92] W.W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 3(8), 1992.

[McC94] W.W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[McC97] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 3(19):263–276, 1997.

[MCK+89] S. Minton, J.G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni, and Y. Gil. Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence*, 40 (Special Volume: Machine Learning), 1989.

[MIL+97] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning*, 18(2):237–246, 1997. Special Issue on the CADE 13 ATP System Competition.

[MKKC86] T.M. Mitchel, R.M. Keller, and S.T. Kedar-Cabelli. Explanation Based Generalization: A Unifying View. *Machine Learning*, 1986.

[Mos96] M. Moser. *Goal-Directed Reasoning in Clausal Logic with Equality*. Number 2 in Dissertationen zur Informatik. CS Press, München, 1996. Accepted as a Ph.D. Thesis at the Fakultät für Informatik, Technische Universität München.

[MOW76] J.D. McCharen, R.A. Overbeek, and L. Wos. Problems and Experiments for and with Automated Theorem-Proving Programs. *IEEE Transactions on Computers*, 25(8):773–782, 1976.

[MR94] S. Muggleton and L. Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[Mug92] S. Muggleton, editor. *Inductive Logic Programming*. Number 38 in The A.P.I.C. Series. Academic Press in association with Turing Institute Press, 1992.

[MW96] E. Melis and J. Whittle. Internal Analogy in Theorem Proving. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 92–105. Springer, 1996.

[MW97a] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[MW97b] E. Melis and J. Whittle. Analogy as a Control Strategy in Theorem Proving. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 367–371. Florida AI Research Society, 1997.

[NN93] P. Nivela and R. Nieuwenhuis. Saturation of First-Order (Constrained) Clauses with the Saturate System . In C. Kirchner, editor, *Proc. of the 5th RTA, Montreal*, number 690 in LNCS, pages 436–440. Springer, 1993.

[NR92] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering Constrained Clauses. In D. Kapur, editor, *Proc. of the 11th CADE, Saratoga Springs*, number 607 in LNAI, pages 477–491. Springer, 1992.

[Qua92] A. Quaife. Automated Deduction in von Neumann-Bernays-Gödel Set Theory. *Journal of Automated Reasoning*, 8(1):91–147, 1992.

[Qui92] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1992.

[Qui96] J.R. Quinlan. Indiction of Decision Trees. *Machine Learning*, 1:81–106, 1996.

[Rei92] W. Reif. The KIV-System: Systematic Construction of Verified Software. In D. Kapur, editor, *Proc. of the 11th CADE, Saratoga Springs*, number 607 in LNAI. Springer, 1992.

[Rei95] W. Reif. The KIV-Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *ORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, number 1009 in LNCS. Springer, 1995.

[RN95] S.J. Russel and P. Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall International, 1995.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Proc. of the 10th Annual Conference on Computer Assurance, Gaithersburg*. IEEE Press, 1995.

[RV99] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 292–2296. Springer, 1999.

[RW69] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[SB99] S. Schulz and F. Brandt. Using Term Space Maps to Capture Search Control Knowledge in Equational Theorem Proving. In A. N. Kumar and I. Russell, editors, *Proc. of the 12th FLAIRS, Orlando*, pages 244–248. AAAI Press, 1999.

[Sch95] S. Schulz. Explanation Based Learning for Distributed Equational Deduction. Diplomarbeit in Informatik, Fachbereich Informatik, Universität Kaiserslautern, 1995.

[Sch96] M. Schramm. *Indifferenz, Unabhängigkeit und maximale Entropie: Eine Wahrscheinlichkeitstheoretische Semantik für nichtmonotones Schließen*. Number 4 in Dissertationen zur Informatik. CS Press, München, 1996. Submitted and accepted as a Ph.D. Thesis at the Fakultät für Philosophie, Logik und Wissenschaftstheorie, Ludwigs-Maximilians-Universität zu München (German Language).

[Sch97] J. Schumann. Automatic Verification of Cryptographic Protocols with SETHEO. In W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 87–11. Springer, 1997.

[Sch98] S. Schulz. Term Space Mapping for DISCOUNT. In J. Denzinger and B. Spencer, editors, *Proc. of the CADE-15 Workshop on* Using AI methods in Deduction*, Lindau*, 1998.

[Sch99a] S. Schulz. *The E Equational Theorem Prover – User Manual*. Automated Reasoning Group, Institut für Informatik, Technische Universität München, 1999. (to appear).

[Sch99b] S. Schulz. System Abstract: E 0.3. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 297–391. Springer, 1999.

[SE90] C.B. Suttner and W. Ertel. Automatic Acquisition of Search Guiding Heuristics. In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, number 449 in LNAI, pages 470–484. Springer, 1990.

[SE91] C.B. Suttner and W. Ertel. Using Back-Propagation for Guiding the Search of a Theorem Prover. *International Journal of Neural Networks Research and Applications*, 2(1):3–16, 1991.

[SF71] J.R. Slagle and C.D. Farrell. Experiments in Automatic Learning for a Multipurpose Heuristic Program. *Communications of the ACM*, 14(2):91–99, 1971.

[SG95] M. Schramm and M. Greiner. Foundations: Indifference, Independence & Maxent. In J. Skilling, editor, *Maximum Entropy and Bayesian Methods in Science and Engeneering (Proc. of the MaxEnt'94)*. Kluwer Academic Publishers, 1995.

[SKG97] S. Schulz, A. Küchler, and C. Goller. Some Experiments on the Applicability of Folding Architecture Networks to Guide Theorem Proving. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 377–381. Florida AI Research Society, 1997.

[SS97a] G. Sutcliffe and C.B. Suttner. Special Issue: The CADE-13 ATP System Competition. *Journal of Automated Reasoning*, 18(2), 1997.

[SS97b] C.B. Suttner and G. Sutcliffe. The TPTP Problem Library (TPTP v2.1.0). Technical Report AR-97-01 (TUM), 97/04 (JCU), Institut für Informatik, Technische Universität München, Munich, Germany/Department of Computer Science, James Cook University, Townsville, Australia, 1997. Jointly published.

[SSY94] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In A. Bundy, editor, *Proc. of the 12th CADE, Nancy*, number 814 in LNAI, pages 252–266. Springer, 1994.

[ST85] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[Sti86] M.E. Stickel. Schubert's Steamroller Problem: Formulation and Solutions. *Journal of Automated Reasoning*, 2(1):89–101, 1986.

[SW49] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[SW99] G. Stenz and A. Wolf. Strategy Selection by Genetic Programming. In A. N. Kumar and I. Russell, editors, *Proc. of the 12th FLAIRS, Orlando*, pages 346–350. AAAI Press, 1999.

[SWL+94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In A. Bundy, editor, *Proc. of the 12th CADE, Nancy*, number 814 in LNAI, pages 341–355. Springer, 1994.

[Tam97] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997. Special Issue on the CADE 13 ATP System Competition.

[Tam98] T. Tammet. Towards Efficient Subsumption. In C. Kirchner and H. Kirchner, editors, *Proc. of the 15th CADE, Lindau*, number 1421 in LNAI, pages 427–441. Springer, 1998.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 2 edition, 1999.

[WAB+99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, G. Jung, E. Keen, C. Theobalt, and D. Topic. System Abstract: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 378–382. Springer, 1999.

[Wei99] C. Weidenbach. Personal communication at CADE-16, Trento. Unpublished, 1999.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER Version 0.42. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 141–145. Springer, 1996.

[WL99] A. Wolf and R. Letz. Strategy Parallelism in Automated Theorem Proving. *International Journal for Pattern Recognition and Artificial Intelligence*, 13(2):219–245, 1999.

[Wol98a] A. Wolf. p-SETHEO: Strategy Parallelism in Automated Theorem Proving. In H. de Swart, editor, *Proc. of the TABLEAUX'98, Oisterwijk*, number 1397 in LNAI, pages 320–324. Springer, 1998.

[Wol98b] A. Wolf. Strategy Selection for Automated Theorem Proving. In F. Giunchiglia, editor, *Proc. of the 8th AIMSA*, number 1480 in LNAI, pages 452–465. Springer, 1998.

[Wol99a] A. Wolf. *Paralleles Theorembeweisen: Leistungssteigerung, Kooperation und Beweistransformation.* Fachreihe Informatik. Herbert Utz Verlag, München, 1999. Accepted as a Ph.D. Thesis at the Institut für Informatik, Technische Universität Münche.

[Wol99b] A. Wolf. Strategy Parallelism: Fast Automated Theorem Proving. In N. Callaos, M. Torres, B. Sanchez, and P. F. Tiako, editors, *Proc. of the World Multiconference on Systemics, Cybernetics and Informatics (SCI-99/ISAS-99)*, volume 2 (Information Systems Development), Skokie, IL, USA, 1999. International Institute of Informatics and Systemics.

[WRC65] L. Wos, G.A. Robinson, and D.F. Carson. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. *Journal of the ACM*, 12(4):536–541, 1965.

[Wus92] J. Wusteman. Explanation-Based Learning: A Survey. *Artificial Intelligence Review*, 6, 1992.

Index