Explanation Based Learning for Distributed Equational Deduction

Stephan Schulz

Diplomarbeit
in
Informatik

Februar 1995

durchgeführt am Fachbereich Informatik
der Universität Kaiserslautern
Betreuer: Prof. Dr. Jürgen Avenhaus und Dr. Jörg Denzinger

Ich erkläre, daß ich die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kaiserslautern, den 28.2.1995

(Stephan Schulz)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automated deduction systems have reached a point where, in some areas, they start to rival human problem solvers. One example is the distributed DISCOUNT system ([AD93], [De93] [Pi92]) for equational reasoning. DISCOUNT can prove many non-trivial theorems faster than a human expert. However, even this sophisticated system breaks down at more difficult and complex problems.

Proof systems can, by now, execute single inferences much faster than any human. The reason for their breakdown in many instances thus cannot be ascribed to a weak reasoning mechanism. However, many experiments have shown that the selection strategies discriminating among the different possible inference steps play a major role for the power of the proof system. This indicates that the selection of good paths through the search space is a fundamental requirement for a potent theorem prover.

Humans generally learn to perform proofs by experience. The ability to find a proof quickly is not automatically present when the necessary calculus is acquired. Good strategies are only discovered when performing or examining proofs in this calculus. In this sense humans *learn* how to find good proofs.

Our analysis of equational proofs (see [Sch93], [DS94a], [DS94b]) has indicated that there is a certain regularity even among proofs from quite different domains. These regularities can be exploited in finding good strategies, using *learning* algorithms to construct good heuristics.

This work develops an explanation-based approach to learning for equational reasoning, with the goal to learn good strategies from previous proofs. The use of the TEAMWORK distribution model allows us to use these learning strategies in cooperation with other, more conventional approaches. This allows for a very powerful and adaptable proof system.

This thesis is structured as follows. After this introduction, the second chapter lists the basic concepts this work is build on. A short survey on machine learning techniques is followed by a section on equational deduction. The TEAMWORK approach for distributed processing is explained, and finally the basics of proof recording and analysis conclude the chapter.

The third chapter analyses the use of knowledge in a TEAMWORK based equational proof system and the possibilities to use learning techniques to improve the knowledge used. Learning to improve the selection strategy for critical pairs is identified as the most promising approach. The next chapter analyses the strong points and the weaknesses of current selection strategies and introduces the basic learning model.

Chapter 5 develops the techniques for the acquisition and management of the knowledge. The structure of the knowledge base is explained. The following two chapters deal with two paradigms for the use of the knowledge: Critical pair evaluation by the reuse of term pair structures and critical pair evaluation with the uses of abstract term evaluation trees. Results for the sequential proof system are given.

Finally, chapter 8 deals with the integration of learning experts into a team. Some promising results are included. Chapter 9 concludes the thesis with a final evaluation and suggestions for further work. The appendices contain problem specifications, an example knowledge base and implementation details.

During my work on this thesis I have been offered a graduate position at the Technische Universität München. I had not expected to be offered a job this early, and thus was forced to complete this thesis in a rush. I hope that the contents did not suffer – I am sure that the presentation did. I hope that occasional spelling errors and grammatical lapses introduced during long nights on coffee and tea do not cripple the intended meaning of the paragraphs. There also are a lot of loose ends in this work. However, I do not think that now is the time to wrap them up – on the contrary, I feel that they offer lanes still to be explored by future research.

Chapter 2

Fundamental Concepts

This chapter introduces basic concepts from the three fields this work is based on. This includes first important concepts from the field of machine learning, followed by an introduction to equational reasoning. Finally, a description of the TEAMWORK method is given, with special consideration of the DISCOUNT system for distributed equational reasoning.

2.1 Machine Learning

The first difficult task encountered when dealing with machine learning techniques is to find a viable definition of *learning*. Learning is a complex feature, and the word is used to describe many at least subtly different activities. If we look at two encyclopedias we find the following relevant definitions of the verb *to learn*:

1. (to) gain knowledge of or skill in by study, experience, or being taught [Ox91]
2. to gain knowledge (of) or skill (in) [DCE81]
3. (to) acquire or develop a particular ability [Ox91]
4. (to) commit to memory [Ox91]
5. to fix in the memory, *memorize* [DCE81]
6. (to) receive instruction; acquire knowledge or skill [Ox91])

Thus, we generally speak of learning if a human acquires some kind of knowledge or skill, usually through interaction with the environment. If he can reproduce previously unknown data or perform a new skill, then that fact is accepted as sufficient proof of a learning process.

A definition of this broadness is not very useful for machines. Computers can store and retrieve great amounts of data trivially. They can also easily be teached (programmed) to perform new operations. In both of these cases the computer is not performing any difficult operations itself. It only stores knowledge presented to it – either raw data or a program.

In order to come to a workable concept we need to look beyond the dictionary definitions. Humans who learn do not simply store the information they are presented with, but automatically restructure it in a way that makes it more usable.

If we transfer this to the machine level we arrive at the following 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.

This definition still requires some elaboration. Just what constitutes an adequate representation of the knowledge?

Learning systems can generally be classified according to their purpose. For systems that only aim at acquiring knowledge there is no easy way to judge the quality of the knowledge representation. Only by detailed analysis and comparison with the concepts the system should learn a conclusion can be reached.

However, many systems use the learning component to improve the performance of an external agent. For these systems we simply call a knowledge representation adequate if this goal is achieved, that is, if the accumulated knowledge enables the agent to perform better.

2.1.1 A Simple Model for Learning

It is possible to describe most learning processes in the framework of a simple model. This model (and much of the presentation on learning systems in general) has been inspired by [RW90]. In a learning situation we can usually distinguish three different components. These are the the *environment*, the *learner* and a *performance element*.

The environment provides data and can give feedback about the quality of the learning. The learner processes the input from the environment, guided by an evaluation of his knowledge representation. Any actor accessing the knowledge of the learner to improve his actions can serve as a performance element. The performance of this actor can be one important element in the evaluation of the knowledge representation. On a higher level performance element and learner are often seen as the same entity.

We can characterize a learning situation using a number of descriptive, representational and functional components:

- The *experience space* `E` is the set of all experiences the learner can use as input. This is usually a language with associated semantics, where each word in the language describes one experience.
- The *hypothesis space* `H` describes all concepts the learner can possibly discover. This is usually some form of knowledge representation language and may or may not be identical to `E`.
- The *knowledge base* `K` stores the knowledge of the learner. It contains both static, precoded knowledge and learned (or *hypothetical*) knowledge subject to change. The hypothetical knowledge usually is represented by a subset of `H`.
- The *experience generator* η uses the knowledge base `K` and interacts with the environment to provide new experiences for the learner.
- The *evaluation function* ν judges the quality of the knowledge representation in `K`. It may use observations of the environment and, particularly, of the performance element.

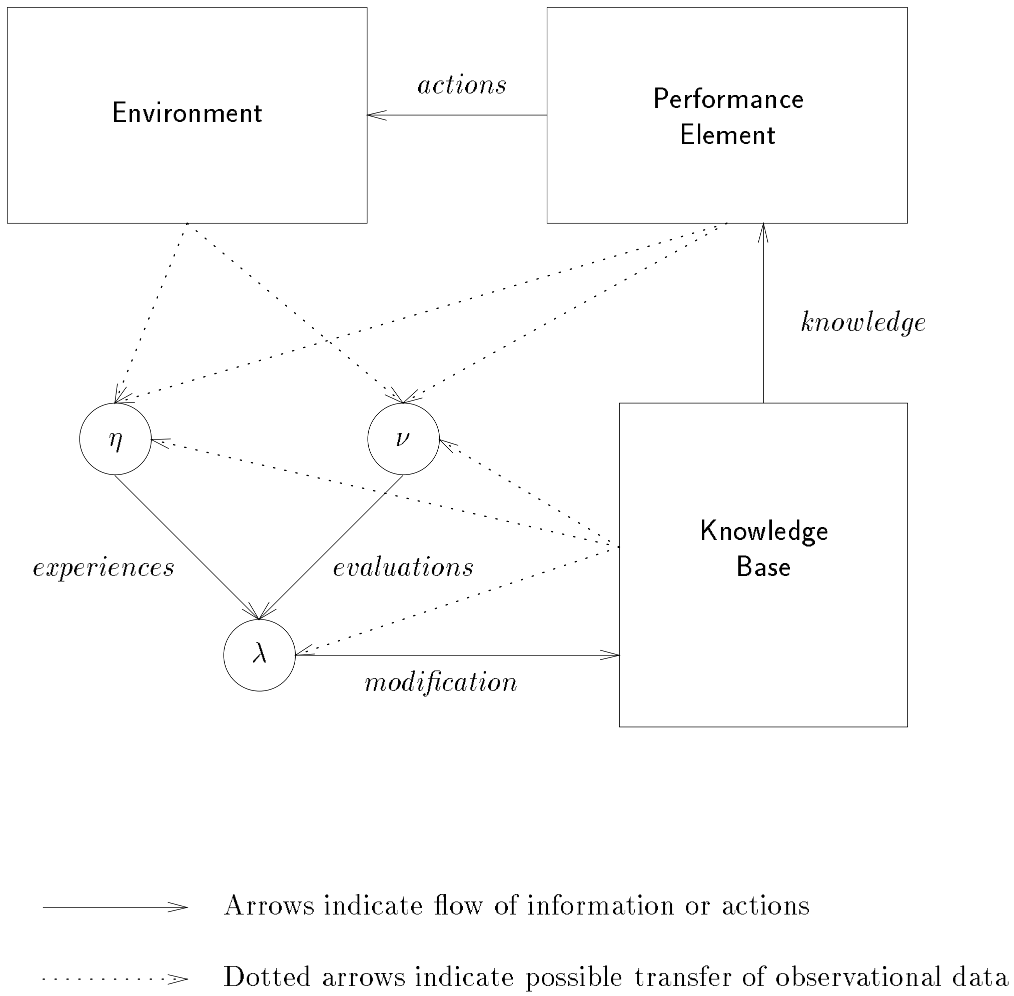


Figure 2.1: A simple learning model

- Finally, the *learning function* λ describes learning inference. The learning function is taking experiences generated by η and evaluations from ν and generates new knowledge in the form of hypothesises. λ is the only element that may modify the knowledge base.

`E` and `H` describe what examples the learner can process and what concepts it can learn. The knowledge base `K` is the very core of the learner. The functional elements η, ν and λ establish the connection between the learner's knowledge base and its environment. They can be of widely varying scope and can be internal to the learner or be considered as part of the environment. Fig. 2.1 shows the different elements and how they relate to each other.

2.1.2 Some Important Learning Situations

Many learning situations can be described by the behavior and placement of the experience generator and the evaluation and learning functions. This section briefly discusses a couple of frequent examples. The next sections will classify more concrete learning paradigms, taking also the knowledge base and experience space into account.

- The learning function λ is the most central functional element of the learner. If λ is fully external we do not speak of learning anymore, but of *programming*. The learner acquires new knowledge, but it is simply entered into the knowledge base.
- Both the experience generator and the evaluation function can be external, while the learning function is part of the learner. In this case we can treat η and ν as a *teacher*.
- In the case of *reinforcement learning*, η generates *no* experiences. λ uses only the evaluation of the performance elements behaviour to improve the knowledge base.
- Another extreme case of learning uses *no* evaluation of the knowledge base. All experiences are simply integrated into the knowledge base using an uniform algorithm. A less extreme case has an (external) evaluation function that is not used to guide the individual learning steps, but only decides whether the goal of the learning process has been reached.
- If the experience generator makes use of the knowledge base to generate new experiences we speak of *closed loop learning*. In *open loop learning*, on the other hand, new experiences are generated only by observing the environment.

2.1.3 Subsymbolic Approaches

Subsymbolic approaches to learning have been a major focus of interest in the last few years. They are not important to this work, however, and will only be mentioned briefly. See [RM86] for a comprehensive introduction and survey.

The subsymbolic approach covers learning systems that do not employ an explicit, symbolic representation of the stored data. The knowledge in these systems is usually represented by the strength of the connections between a large number of simple, independent units. Often these systems are called *neural networks*, because they are inspired by, and sometimes even try to emulate, the interconnected patterns of neurons in human or animal brains.

The single processing units, often called *neurons*, can be *input units*, *output units* or *hidden units*. Input units are simply set to a certain value from the outside, output units produce the result of the computation and hidden units are invisible from the outside and only perform calculations. Depending on the exact structure of the net, units can be both input and output units.

The neurons are connected to each other. A neuron receives input from a number of other units, performs a simple calculation on the input, and produces an output value that is propagated to further units. Neural nets usually learn by changing the strengths of the connections according to one of several *learning rules*.

We can consider neural networks within the frame of the learning model from the previous section. In the most frequent case of (multilevel) feed-forward nets (often called *perceptrons*), an *experience* consists of an pattern clamped to the input neurons. The values are then propagated through the net, and the resulting output is compared to the desired output. This *evaluation* is then used by the *learning rule* to update the connections. In this case both the experiences and the evaluation are provided by an outside teacher.

Another approach uses an internal evaluation function. Randomly introduced biases (represented by the initial connections) are strengthened if they distinguish between examples displaying different features. *Competitive learning* is one example for this concept. Experiences (input patterns) are still presented by an external teacher, but the system acts as a *feature detector* and classifies the examples according to self-found criteria.

Neural nets have proved to be a good choice for pattern recognition. They also are very suitable for the processing of analogue input data from simple sensors, and they are often able to discover previously unknown relationships in the input data. They are highly fault tolerant and can generalize beyond the scope of the presented examples.

However, neural nets are very expensive to simulate on a sequential architecture (or a distributed architecture with few processors or low connection bandwidth). They also need a very large number of examples to discover even quite simple relationships.

In addition to that, complex relationships or even complex input patterns usually require a large number of neurons. This further slows down the learning process.

If the input data already is available in a symbolic form, if complex relationships are involved and if a single example can be expensive to generate, neural networks are only the second choice. Symbolic approaches often give better and faster results.

2.1.4 Synthetic Learning

Synthetic learning algorithms try to *synthesize* general concepts from a large number of instances. They are usually given a large set of preclassified examples. The algorithm tries to fit a concept description to these examples. This involves *inductive reasoning*[1].

There are a couple of different approaches to this.

- The most simple approach is to find the least general generalization of the examples for each class (using an *anti-unification* algorithm). A pure anti-unification algorithm (see for example [Ri89]) cannot cope with *noisy data* (noisy data may contain misclassified examples) and cannot benefit from negative instances of the concept.

- The *version space algorithm* is more sophisticated. It maintains a set of concept descriptions that are both correct and complete with respect to the examples given. Positive examples are used to exclude too narrow concepts, negative examples are used to exclude too general concepts. The algorithm only works for concepts that can be expressed as conjunctions of simple properties, and it cannot handle noisy data. However, it is guaranteed to find a correct solution if these conditions are fulfilled and enough examples are presented.

- The *AQ algorithm* ([ML78], [CN89] also gives a sufficient introduction) extends the version space method in order to learn disjunctive concepts. This is achieved by attempting to learn each element of the disjunction separately. The algorithm produces a set of rules for the classification of new examples.

- The *CN2 algorithm* ([CN89]) is similar to the AQ algorithm, but generates only heuristic

[1] Induction is used here in the classical sense of a generalization from few instances to a general rule or concept, not in the sense of mathematical induction.

rules, thus coping with noisy data. It attempts to find *best*[2] premises, and generates a rule that maps all instances of a premise onto the class most frequently encountered among the training examples that are instances of that premise.

- Finally, *decision tree algorithms* (see e.g. [Qu86]) use the examples to build decision trees. In these trees a node corresponds to a subset of possible classes and a link represents a predicate on the examples. The premise for a class is described by a path from the root to a leaf, and the leaf corresponds to the class of the object. The optimal tree constructed by a learning algorithm minimizes the expectation value of the path length.

Using the general model, the experiences consist of preclassified examples (usually presented by a teacher). The learning function generates fitting concept descriptions and the evaluation function judges the quality of the representation either internally, using probabilistic approaches, or according to the success of the system in classifying new examples.

2.1.5 Analytical and Explanation-Based Learning

The algorithms presented so far use a large number of examples and try to find a common definition. They are searching for inter-example relationships. Analytical learning, on the other hand, searches for the important properties of a single object in order to acquire new knowledge. These *intra-example* properties are then generalized to allow conclusions about other examples.

In order to exploit properties of single examples, very much domain knowledge is required. Consequently, analytical methods do not need large numbers of experiences, but are very *knowledge intensive*. They are usually applied to improve the performance of an agent – either a classifier or, more often, a proof or planning system. Almost all analytical learning algorithms can be described as variations of *explanation-based learning (EBL)*. [Wu92] provides a good overview over this approach to learning.

As stated above analytical learning can only detect intra-example properties. No new domain knowledge about the objects is gathered. The existing knowledge is only reorganized in a more *operational* form. Analytical (or, more concrete, explanation-based) learning is therefore usually employed in connection with a performance element – a proof system, a planning system or an arbitrary inference engine used for complex computations. The aim of explanation-based reasoning is to find and record shortcuts through the search space - either by direct generation of macro operations or by acquiring *control knowledge*.

A single explanation-based learning step works as follows: The performance element is given a goal. Using extensive domain knowledge, the goal is reached by performing a sequence of inferences (or applying a number of abstract *operators*). This sequence is analyzed, and an *explanation* why this particular sequence achieves the goal is extracted. This explanation (or the sequence itself) is *generalized* as much as possible (or feasible). The learned concept is this generalized explanation. It can be used to aid future reasoning.

There are different approaches to the explanations and their generalization. Many systems learn from *positive* examples only, others incorporate negative examples as well. Some Authors allow only generalizations that ensure that the generalized explanation still is valid

[2] *Best* premises are premises that include a maximal number of examples from one class and as few examples from other classes as possible.

within the domain theory. This approach is often labeled *explanation-based generalization (EBG)* ([MKK86] – however, the literature is far from consistent in this respect). Other approaches use over-generalization techniques to learn heuristic rules – see for example [MC+89] and [Bh90].

In order to be versatile, the performance element needs to accumulate a decent number of explanations. Thus, while a learning step only takes a singular experience, the power of the paradigm comes from the accumulation of many examples. This makes explanation-based reasoning a close relative to procedures like *derivational analogy* and *case-based reasoning.* In fact, in many modern approaches these methods are nearly indistinguishable.

2.1.6 Case- and Analogy-Based Learning

Case- and analogy based procedures are inspired by a simple observation: Similar objects often display similar behaviour, and similar problems have similar solutions. Thus previous experiences can be used to aid reasoning about new, similar examples. In order to make this possible, two elements are necessary. The first one is a database of previous examples (or *cases*), the second is a *similarity measure* used to locate an appropriate previous case.

The case base is the part of the learner that changes during the learning process. It stores new experiences, guided by an evaluation function. Many systems simply store *all* previous cases, other systems try to limit the size of the case base by including only cases necessary to establish a sufficient coverage of the experience space. Generally the cases are stored exactly as they are generated – this is the main difference to explanation-based approaches. However, the transition is a steady one, and many case based systems generalize experiences, too.

The similarity measure is used to determine the *nearest neighbour* in the case base, the previous example that resembles the new case best. This case is then used to handle the new problem. Classification problems are usually solved by simply presuming the class of the nearest neighbour. For more complex problems, though, complex inferences may be needed.

In *analogical reasoning* the similarity is established by an analogy mapping, mapping features of the old case onto the new case. Simple mappings designate similar cases. If a case is selected this mapping can also be used to map the solution of the old case onto the new problem. Kolodner in [Ko87] classifies case-based systems according to the different mappings possible:

- In the most simple version the solution of the old case is simply transfered.
- A more involved version does not only transfer the solution, but also modifies it wherever it is necessary.
- A rather different approach does not try to transfer the old solution, but to adapt the *method* the old solution was found by.
- Finally, it is possible to create an *abstraction* of the old case and try to apply this to the new case. Note that this also describes explanation-based generalization quite well.

The new solution is not necessarily correct – the analogy does not always hold. The solution has to be checked by the problem solver, either by using the data available from the old case or by an external evaluator.

2.2 Equational Deduction

This section introduces a number of concepts on equational reasoning. I will not give a complete overview of the field, but concentrate on the topics important for an unfailing completion based prover. Also, as this chapter is mainly intended to clarify the methods and notations used, I will omit all proofs.

For a more in-depth introduction to the field, using similar notations, see [Av91] and [Av90]. [Sch93] contains a shorter introduction, including some of the theoretical background omitted here.

An equational theorem prover is a system trying to deal with the following problem:

> Given a set `E` of equations (over terms), is an equation $s{=}t$ a logical consequence of `E` (written as $s =_{\mathtt{E}} t$)?

In order to allow a more detailed discussion of the related problems we need some basic definitions.

2.2.1 Notations and Basic Definitions

- A *term* t is a recursive structure build from a set F of function symbols and a set V of variables. The set of all terms for given sets F and V is called $Term(F,V)$. A *ground term* is a term not containing any variables, the set of ground terms is denoted as $Term(F)$.
- We write $t|_p$ to denote the subterm of t at the position p. The top position is written as λ, and $t|_{p.q} \equiv (t|_p)|_q$.
- A substitution σ is a function mapping a finite set of variables onto the set of terms. We write $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ if σ maps x_i to t_i for $1 \leq i \leq n$. Substitutions can be applied to complex terms by applying them to the variables occuring in these terms.
- σ_{id} represents the empty substitution $\{\}$. A *unifier* for two terms is a substitution that maps both terms to a common specialization. The *most general unifier* for two terms t_1 and t_2 is denoted by $mgu(t_1, t_2)$.
- The *encompassment ordering* $\trianglerighteq$ is defined by $s \trianglerighteq t$ iff $\sigma(s) = t|_p$ for a substitution σ and a position p. $\triangleright$ is the strict part of $\trianglerighteq$.
- $t[p \leftarrow t']$ denotes the term t with the subterm at position p replaced by t'.
- An equation is a pair $(s,t) \in (Term(F,V) \times Term(F,V))$. We write $s{=}t$ instead of (s,t). We always regard equations as symmetrical, so that $s{=}t$ also represents $t{=}s$.
- A *reduction ordering* $>$ is a Noetherian ordering compatible with the term structure and stable with respect to substitutions. A reduction ordering total on ground terms is called a *ground reduction ordering*.
- A rule is a pair $(l,r) \in (Term(F,V) \times Term(F,V))$ with $Var(r) \subset Var(l)$ ($Var(t)$ denotes the set of variables occurring in t). We write a rule as $l{\rightarrow}r$.

- A rule $l \rightarrow r$ is *compatible* with a reduction ordering $>$ if $l > r$. A system of rules $\mathtt{R}$ is compatible with $>$ if all rules in $\mathtt{R}$ are compatible.

An equation $s{=}t$ is a logical conclusion from a set of equation $\mathtt{E}$ if it is valid in each model of $\mathtt{E}$. According to Birkhoff's theorem $s =_{\mathrm{E}} t$ holds, if and only if we can transform s into t by application of the equations from $\mathtt{E}$. Given this result we can give an *operational* characterization of $\mathtt{E}$-equality.

- Given a set $\mathtt{E}$ of equations we define a symmetrical relation $\vdash\!\dashv_{\mathrm{E}}$ as follows:

  $t_1 \vdash\!\dashv_{\mathrm{E}} t_2$ iff there exists an equation $s{=}t \in \mathtt{E}$, a position p in t_1 and a substitution σ with $\sigma(s) \equiv t_1|_p$ and $t_2 \equiv t_1[p \leftarrow \sigma(t)]$.

  A $\vdash\!\dashv_{\mathrm{E}}$-step is also called an *application of* $s{=}t$. Then $=_{\mathrm{E}}$ is the reflexive and transitive closure of $\vdash\!\dashv_{\mathrm{E}}$, that is $t_1 =_{\mathrm{E}} t_2$ if and only if $t_1 \overset{*}{\vdash\!\dashv}_{\mathrm{E}} t_2$.

Knuth-Bendix completion (see [KB70]) tries to substitute the application of equations with the application of rules (oriented equations). It tries to generate a confluent rule system by orienting equations according to a reduction ordering, generating new equations from *critical pairs* and using rules to simplify the knowledge base.

In order to handle unorientable equations the calculus introduced by Knuth and Bendix has been extended in [BDP89]. The extended calculus – unfailing Knuth-Bendix completion – tries to generate only a *ground confluent* system, and orientable instances of equations can be used for simplifications.

- A system $\mathtt{R}$ of rules defines a rewriting relation $\Longrightarrow_R$ as follows:

  $t_1 \Longrightarrow_R t_2$ iff there exists a rule $l \rightarrow r$, a place p in t_1 and a substitution so that $t_1|_p \equiv \sigma(l)$ and $t_2 \equiv t_1[p \leftarrow \sigma(r)]$

  If $\mathtt{R}$ is compatible with a reduction ordering, the relation $\Longrightarrow_R$ is guaranteed to be Noetherian.

- A relation $\longrightarrow$ is called *confluent* if each divergence possible in $\longrightarrow$ can be rejoined.

- A term that cannot be reduced with $\Longrightarrow_R$ is said to be in *normal form* with respect to $\mathtt{R}$.

- The set $\mathtt{R}_{\mathtt{E}} = \{\sigma(s) \rightarrow \sigma(t) | \sigma(s) > \sigma(t), s{=}t \in \mathtt{E}, \sigma \text{ a substitution}\}$ is called the set of *orientable instances* for a set of equations $\mathtt{E}$ and a reduction ordering $>$.

- We define $\mathtt{R}(\mathtt{E}) = \mathtt{R} \cup \mathtt{R}_{\mathtt{E}}$ for given sets $\mathtt{E}$, $\mathtt{R}$ and an ordering $>$.

- *Critical pairs* are defined as follows: Let $l_1{=}r_1$ and $l_2{=}r_2$ be two equations and let $>$ be a (ground) reduction ordering. Let p be a non-variable position in l_1, $\sigma = \mathtt{mgu}(l_1|_p, l_2)$. If $\sigma(l_1) \not< \sigma(r_1)$ and $\sigma(l_2) \not< \sigma(r_2)$ then $\langle \sigma(r_1), \sigma(l_1[p \leftarrow r_2]) \rangle$ is called a critical pair between the two equations at position p.

- Finally, critical pairs between rules can be built by treating rules just like equations. If the rules are compatible with $>$, only the left hand sides have to be overlapped. The set of all critical pairs that can be build for an $\mathtt{E}$ and $\mathtt{R}$ is called $\mathtt{CP}(\mathtt{E}, \mathtt{R})$.

2.2.2 Completion based proof procedures

Input for a completion based proof system are a set of equations E, a (ground) reduction ordering $>$ and a (skolemized) goal $s{=}t$. In order to prove $s =_{\mathrm{E}} t$ the prover successively orients equations from E, creating a set R of rules, generates new equations by building critical pairs between rules and equations from E and R and uses the rules from R and orientable instances from equations for simplifications of both the goal and the knowledge base. This process is called *completion*. The goal is proved if the normal forms of both sides (with respect to R(E)) are identical or if an equation subsuming the goal is being generated.

In order to guarantee completeness and correctness of the prover we need to discuss completion more formally. The authors in [BDP89] suggested an inference system describing the operations on the rules and equations. We use this inference system, with an added rule to allow for subsumption. If these inference rules are applied using a *fair* strategy, the resulting prover can be shown to be both complete and correct.

Definition 1 : The inference system U (Unfailing Completion)
Let $>$ be a ground reduction ordering. The inference system works on pairs (E, R), where E is a set of equations and R is a set of rules compatible with $>$.

(U1) Orient an equation

$$\frac{(\mathtt{E} \cup \{s{=}t\}, \mathtt{R})}{(\mathtt{E}, \mathtt{R} \cup \{s{\to}t\})} \qquad \text{if } s > t$$

(U2) Generate an equation

$$\frac{(\mathtt{E}, \mathtt{R})}{(\mathtt{E} \cup \{s{=}t\}, \mathtt{R})} \qquad \text{if } s \leftarrow\!\!\dashv_{\mathtt{E}\cup\mathtt{R}} u \vdash\!\!\rightarrow_{\mathtt{E}\cup\mathtt{R}} t,\ s \not> u,\ t \not> u$$

(U3) Simplify an equation

(a) $$\frac{(\mathtt{E} \cup \{s{=}t\}, \mathtt{R})}{(\mathtt{E} \cup \{u{=}t\}, \mathtt{R})} \qquad \text{if } s \Longrightarrow_{R} u$$

(b) $$\frac{(\mathtt{E} \cup \{s{=}t\}, \mathtt{R})}{(\mathtt{E} \cup \{u{=}t\}, \mathtt{R})} \qquad \text{iff } s \Longrightarrow_{R_E} u \text{ using a rule } l{\to}r \text{ with } s \rhd l$$

(U4a) Delete an equation

$$\frac{(\mathtt{E} \cup \{s{=}s\}, \mathtt{R})}{(\mathtt{E}, \mathtt{R})}$$

(U4b) Subsume an equation

$$\frac{(\mathtt{E} \cup \{s{=}t, u{=}v\}, \mathtt{R})}{(\mathtt{E} \cup \{s{=}t\}, \mathtt{R})} \qquad \text{if } u|_p \equiv \sigma(s), v \equiv [p \leftarrow t],\ u \rhd s, \text{ for a position } p \text{ and a substitution } \sigma$$

(S1) Simplify the right side of a rule

$$\frac{(\mathtt{E}, \mathtt{R} \cup \{s{\to}t\})}{(\mathtt{E}, \mathtt{R} \cup \{s{\to}u\})} \qquad \text{if } t \Longrightarrow_{R(E)} u$$

(S2) Simplify the left side of a rule

$$\frac{(\mathtt{E}, \mathtt{R} \cup \{s{\to}t\})}{(\mathtt{E} \cup \{u{=}t\}, \mathtt{R})} \qquad \text{if } s \Longrightarrow_{R(E)} t \text{ using a rule } l{\to}r \text{ with } s \rhd l$$

We write $(\mathtt{E},\mathtt{R}) \vdash_{\mathtt{U}} (\mathtt{E}',\mathtt{R}')$ if $(\mathtt{E},\mathtt{R})$ can be transformed into $(\mathtt{E}',\mathtt{R}')$ using one inference rule from **U**. An application of an inference rule from **U** does not change the equality relation described by the system:

Theorem 1 : Correctness of U
Assume $(\mathtt{E},\mathtt{R}) \vdash_{\mathtt{U}} (\mathtt{E}',\mathtt{R}')$ and consider $\mathtt{R}$ to be compatible with a reduction ordering $>$. The the following assertions hold:

- $\mathtt{R}'$ is compatible with $>$.
- $=_{\mathtt{E}\cup\mathtt{R}} \;=\; =_{\mathtt{E}'\cup\mathtt{R}'}$.

A proof for an equation $s{=}t$ in $\mathtt{E} \cup \mathtt{R}$ is a chain of terms connected by one application of a rule or equation from $\mathtt{E} \cup \mathtt{R}$. Using the ground reduction ordering $>$ and the encompassment ordering $\rhd$ a well founded proof ordering $>_p$ that is compatible with the proof structure can be constructed (see [BDP89]), such that the following theorem holds:

Theorem 2 : Proof orderings
Let $>_p$ be a proof ordering, and assume $(\mathtt{E},\mathtt{R}) \vdash_{\mathtt{U}} (\mathtt{E}',\mathtt{R}')$. Let $\mathtt{B}$ be a proof for $s{=}t$ in $\mathtt{E} \cup \mathtt{R}$. Then there exists a proof $\mathtt{B}'$ for $s{=}t$ in $\mathtt{E}' \cup \mathtt{R}'$ with $\mathtt{B} \geq_{\mathtt{p}} \mathtt{B}'$. In particular, if $\mathtt{B}$ contains a peak $s \Longleftarrow_{R(E)} u \Longrightarrow_{R(E)} t$ and $\mathtt{B}'$ is constructed from $\mathtt{B}$ by replacing this peak with a new equation generated using rule (U2), then $B >_p B'$.

If only minimal (with respect to $>_p$) proofs are considered and any peak in these is eventually eliminated by generating new equations, it can be assured that an equivalent proof containing no peaks (a so called V-proof) does exist for every ground proof of a valid equation. To ensure that every peak is eliminated we have to make certain demands on the sequence of the inferences.

Definition 2 : U-fairness

- A sequence $(\mathtt{E}_\mathtt{i},\mathtt{R}_\mathtt{i})_{\mathtt{i}\in\mathbb{N}}$ is called an **U**-derivation if $(\mathtt{E}_\mathtt{i},\mathtt{R}_\mathtt{i}) \vdash_{\mathtt{U}} (\mathtt{E}_{\mathtt{i+1}},\mathtt{R}_{\mathtt{i+1}})$ for all $\mathtt{i} \in \mathbb{N}$.
- The system $(\mathtt{R}^\infty,\mathtt{E}^\infty)$ of *persistent* rules and equations (for a given **U**-derivation) is defined by:
$$\mathtt{E}^\infty = \bigcup_{\mathtt{i}\geq 0}\bigcap_{\mathtt{j}\geq \mathtt{i}} \mathtt{E}_\mathtt{j} \qquad \text{and} \qquad \mathtt{R}^\infty = \bigcup_{\mathtt{i}\geq 0}\bigcap_{\mathtt{j}\geq \mathtt{i}} \mathtt{R}_\mathtt{j}$$
- An **U**-derivation $(\mathtt{E}_\mathtt{i},\mathtt{R}_\mathtt{i})_{\mathtt{i}\in\mathbb{N}}$ is fair, if
$$\mathtt{CP}(\mathtt{E}^\infty,\mathtt{R}^\infty) \subseteq \bigcup_{\mathtt{i}\geq 0} \mathtt{E}_\mathtt{i}$$

A derivation is fair, if every critical pair between persistent rules will eventually be built and added to the set of equations. It can be shown that the following theorem holds for fair **U**-derivations.

Theorem 3 : Completeness of fair U-derivations
Let $(\mathtt{E}_\mathtt{i},\mathtt{R}_\mathtt{i})_{\mathtt{i}\in\mathbb{N}}$ be a fair **U**-derivation.

- The final system of rules and equations describes the same equality as the initial system: $=_{\mathtt{E}^\infty\cup\mathtt{R}^\infty} \;=\; =_{\mathtt{E}_0\cup\mathtt{R}_0}$.

- If $s =_{\mathrm{E}} t$ holds, then there is an i such that the normal forms of s and t with respect to $(\mathtt{E}_i, \mathtt{R}_i)$ are identical.

This result can be used to build completion based provers for equational logic. The following algorithm describes a possible implementation of such a proof system. It will either proof a given goal or it will try to generate a (possibly infinite) ground confluent system of rules and equations that can be used for calculations in equationally specified algebraic structures.

In order to keep track of the critical pairs already processed it uses not two, but three sets of term pairs to represent the current state of a completion process: A set `E` of processed, but unorientable equations, a set `R` of rules (processed and oriented equations) and a set `CP` of unprocessed equations.

The completion algorithm starts out with empty sets `R` and `E`, and with the initial axioms in `CP`. It examines each equation in `CP`, reduces it to normal form with respect to `E` and `R`, uses it to build new critical pairs (to be added to `CP`) and to eliminate redundancies from `R` and `E` by simplification (this process is known as *interreduction*). It is then added to either `R` (if it can be oriented according to $>$) or `E`.

To build a prover on top of the completion algorithm the goal is brought to normal form with respect to each successive `E` and `R`. If these normal forms are identical or if an equation from `E` subsumes the goal, the goal is proved. Please note that both, completeness and efficiency of the proof process, depend on the order in which the equations from `CP` will be considered, with both goals often conflicting.

An Algorithm for Equational Deduction

The following algorithm uses unfailing Knuth-Bendix completion to either generate a ground confluent set of rules and equations or to proof a given goal. It is a simplified sketch of the algorithm actually implemented in DISCOUNT.

Input:	`M`	A set of equations
	$>$	A ground reduction ordering
	`PROOFMODE`	A boolean value, `TRUE`, if a single goal is to be proved, `FALSE` if a ground confluent end system is desired
	`(gs,gt)`	The (skolemized) goal to be proved (only if `PROOFMODE` has the value `TRUE`
Variables:	`R`	The set of processed rules
	`E`	The set of processed equations
	`CP`	The set of unprocessed equations
	`s,t`	The sides of the unprocessed term pair considered at a given moment
	`l,r`	The terms of a newly generated rule
	`u,v`	The sides of a processed term pair considered again
	`u',v'`	Possibly simplified descendants from `u` and `v`
Functions:	`NOTEMPTY(list)`	FALSE, if `list` is empty, TRUE otherwise
	`FIRST(list)`	First element of `list`
	`EXCEPTFIRST(list)`	`list` without its first element
	`NORMALFORM(t,R)`	Calculates a normal form for `t` with respect to `R`
	`SUBSUM(e,E)`	Tests, whether the equation `e` is subsumed by an equation in `E` or whether an equivalent equation already exists in `E`
	`INSERT(list,e)`	Inserts the term pair `e` into `list` in a way that ensures that for any given pair in `list` only finitely many pairs will be inserted in front of it (this ensures fairness). This is usually realized by evaluating the critical pair using some heuristic and inserting it at the appropriate position in the list
	`CPS(e,E)`	Returns all critical pairs that can be build between `e` and term pairs from `E`

```
BEGIN
  CP := M;
  E := {};
  R := {};
  WHILE NOTEMPTY(CP)
    IF PROOFMODE = TRUE THEN
      gs := NORMALFORM(gs,R(E));
      gt := NORMALFORM(gt,R(E));
      IF gs = gt THEN END; A proof has been found
    ENDIF
    (s,t) := FIRST(CP);
    CP := EXCEPTFIRST(CP);
    s := NORMALFORM(s,R(E));
    t := NORMALFORM(t,R(E));
    IF s≠t AND NOT(SUBSUM(s=t,E)) THEN
      FOREACH (u,v)∈ E
        u' := NORMALFORM(u,R_{s=t});
        v' := NORMALFORM(v,R_{s=t});
        IF u≠u' OR v≠v' THEN
          E := E \ {u=v};
          CP:= INSERT(CP,u'=v');
        ENDIF
      ENDFOREACH
      FOREACH (u,v)∈ R
        v := NORMALFORM(v,R(E ∪ {s = t}));
        u' := NORMALFORM(u,R_{s=t});
        IF u'≠u THEN
          R := R\{(u,v)};
          CP := INSERT(CP,u'=v);
        ENDIF
      ENDFOREACH
      FOREACH (u,v) ∈ CPS(s=t,E ∪ R)
        CP := INSERT(CP,u=v);
      IF s>t THEN
        R := R ∪ {s→t};
      ELSE IF t>s THEN
        R := R ∪ {t→s};
      ELSE
        E:= E ∪ {s=t};
      ENDIF
    ENDIF
  ENDWHILE
  (E,R) represents the ground confluent end system
END
```

As the algorithm always picks the first critical pair from `CP`, the direction of the completion process is controlled by the function `INSERT`. We will take a closer look at a possible realization of this function:

Input:	`list`	A list of equations
	`e`	A single equation to be inserted into list
Functions[3]:	`EVAL_PAIR(e)`	A function mapping an equation to an integer number greater to or equal to a base value (hopefully realizing a quality measure designating the importance of the pair for the proof process – a low number means the pair is important)

```
INSERT(list,e)
  IF EVAL_PAIR(e) > EVAL_PAIR(FIRST(list)) THEN
    RETURN FIRST(list).INSERT(EXCEPTFIRST(list,e));
  ELSE
    RETURN e.list;
  ENDIF;
END
```

The actual implementation in the DISCOUNT system conducts the linear search iteratively and uses a sophisticated caching algorithm to speed it up. It also calculates the weights once and stores them with the pairs. However, it performs basically just like the simple sketch shown above. Please note that the knowledge used to guide the completion is completely provided by the function `EVAL_PAIR`. With these assumptions about the algorithm we can conclude the section with the following theorem:

Theorem 4 : Fairness of the completion algorithm
The completion algorithm described in this section generates an **U**-fair derivation, if the sets $\{\mathtt{e} | \mathtt{e} \in (Term(F,V) \times Term(F,V)), \mathtt{EVAL\_PAIR(e)} = i\}$ are finite for each $i \in \mathbf{N}$.[4]

Proof: According to definition 2, a derivation is **U**-fair, if all critical pairs between persistent rules and equations are being built. In the algorithm, all critical pairs between rules and equations from `R` and `E` have been built. It remains to be shown that no equation remains in `CP` forever (if it is removed, it is not persistent, if it is moved to `E` or `R`, its critical pairs will be generated).

Now assume $\mathtt{cp} \in \mathtt{CP}$ with $\mathtt{EVAL\_PAIR(cp)} = \mathtt{j}$. According to the definition of `EVAL_PAIR`, only integers greater than or equal to a base value are generated as weights. Thus, there are only finitely many weights i possible that are smaller than or equal to j. But we assume that for each i only a finite number of pairs is mapped onto i. Thus, only a finite number of pairs will ever be inserted before `e`. As each execution of the main loop of the algorithm removes the first pair form `CP`, `e` is eventually reached and also removed.

[3] For explanations of the other used functions look at the previous algorithm.

[4] Please note that this is a sufficient, not a required condition. It is also sufficient to ensure *team fairness* (Compare [AD93]).

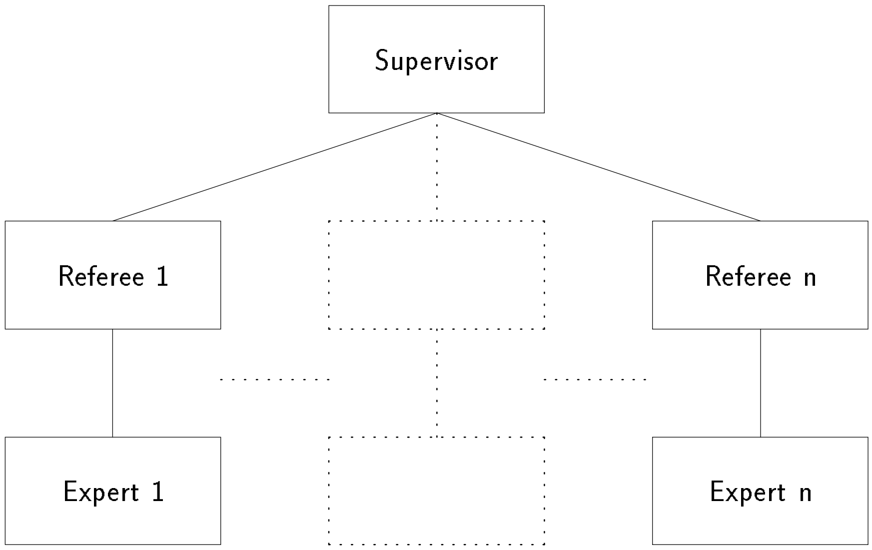


Figure 2.2: Basic structure of a team in the TEAMWORK method

2.3 Distributed Deduction - The TEAMWORK Approach

The TEAMWORK method (see [De93] or [AD93]) is an approach to distribute theorem proving procedures. It has been inspired by human project teams. A TEAMWORK based deduction system models human project teams by use of multiple processes running on different processors. However, the aim is not only to utilize the additional power of multiple processors, but to get synergetic effects by following more than one threads at a time.

A *team* consists of a single *supervisor* and a number of *experts*, each accompanied by a *referee* evaluating his work. The basic structure is depicted in Fig. 2.2. Most of the actual problem solving is done by the experts. Each expert is usually working without communicating with the other team members. Only at *team meetings* scheduled by the supervisor results are exchanged.

The supervisor is responsible for the composition of the team. He selects the experts to work on a specific task initially by judging their previous success on related problems, later by using the referees' evaluation of their performance in dealing with the problem at hand. He is also assigning a referee to each expert.

The referees are evaluating the work of the different experts. Their conclusions are used to change the composition of the team, to find the most promising system among the work by the experts, and to find important results (from their respective experts) for exchange at the team meetings.

The experts are the members of the team working directly on the problem. In our case each of them is using an unfailing completion algorithm[5] as described in the previous chapter. They differ only in the criteria used to select inferences.

[5]This is a slight simplification. We can also add *specialists* using other algorithms. One example is a *reduction specialist* trying to eliminate critical pairs before they are selected by the other experts.

At the team meetings the system of the best expert is chosen as the base for further work (In our case of unfailing completion, and in this context, the term *system* refers to the three sets `E`, `R` and `CP` from the algorithm in section 2.2.2). As only this one system survives completely, the experts are competing for the best result.

However, competition is only one aspect of the TEAMWORK method. The second important element is the *cooperation* between the experts. Cooperation is achieved by integrating outstanding results from inferior experts (as chosen by their referees) into the system of the best expert. This is done by the supervisor during the team meetings, before he presents a new and updated problem description to the experts for the next working phase (In our case of a completion based theorem prover the exchanged results – rules and equations – are handled as new critical pairs to be processed immediately).

It is important to note that most of the results generated by the inferior experts are dropped or *forgotten*. We believe that one of the reasons for the success of TEAMWORK is this feature of forgetting, because it alleviates the usual explosion of the search space. Our experiments have, up to now, supported this view.

Fig. 2.3 shows an example for a complete cycle, including two team meetings, for a team of 3 experts. It can be described as follows:

- The supervisor selects a team of experts with their respective referees. Each of the experts is given the problem description. The supervisor schedules the next team meeting.
- The experts start working on the problem. If none of them can solve it until the scheduled meeting their progress is evaluated by the respective referees. The referees also select outstanding single results and report these results and their overall evaluation of the experts progress to the supervisor.
- The supervisor chooses the best expert and integrates the results from the other experts into this expert's system. He chooses a new team based on the success of the experts in the previous working phases. The experts are handed the updated problem description and the cycle starts again.

It should be noted that experts, referees and the supervisor work at different times. It is therefore feasible to use a single process on each computing node to represent all three entities. Thus, after the working cycle the experts simply switch to referee mode, and the referees can immediately access the experts' data structures. By using floating control even the supervisor can reside on different nodes. Using this mechanism cuts down on the (expensive) network traffic. Most of the communication becomes *virtual*.

In [De93] and [AD93] the authors proved that a completion based prover using TEAMWORK is complete, if certain (weak) fairness criteria are fulfilled.

2.4 Proof Recording and Analysis

In order to understand proofs and provers, and in order to learn more about them, the work of the provers and the proofs have to be available in some form. Our previous research in this area (see [Sch93], [DS94b]) indicated that, at least for completion based proof systems, an

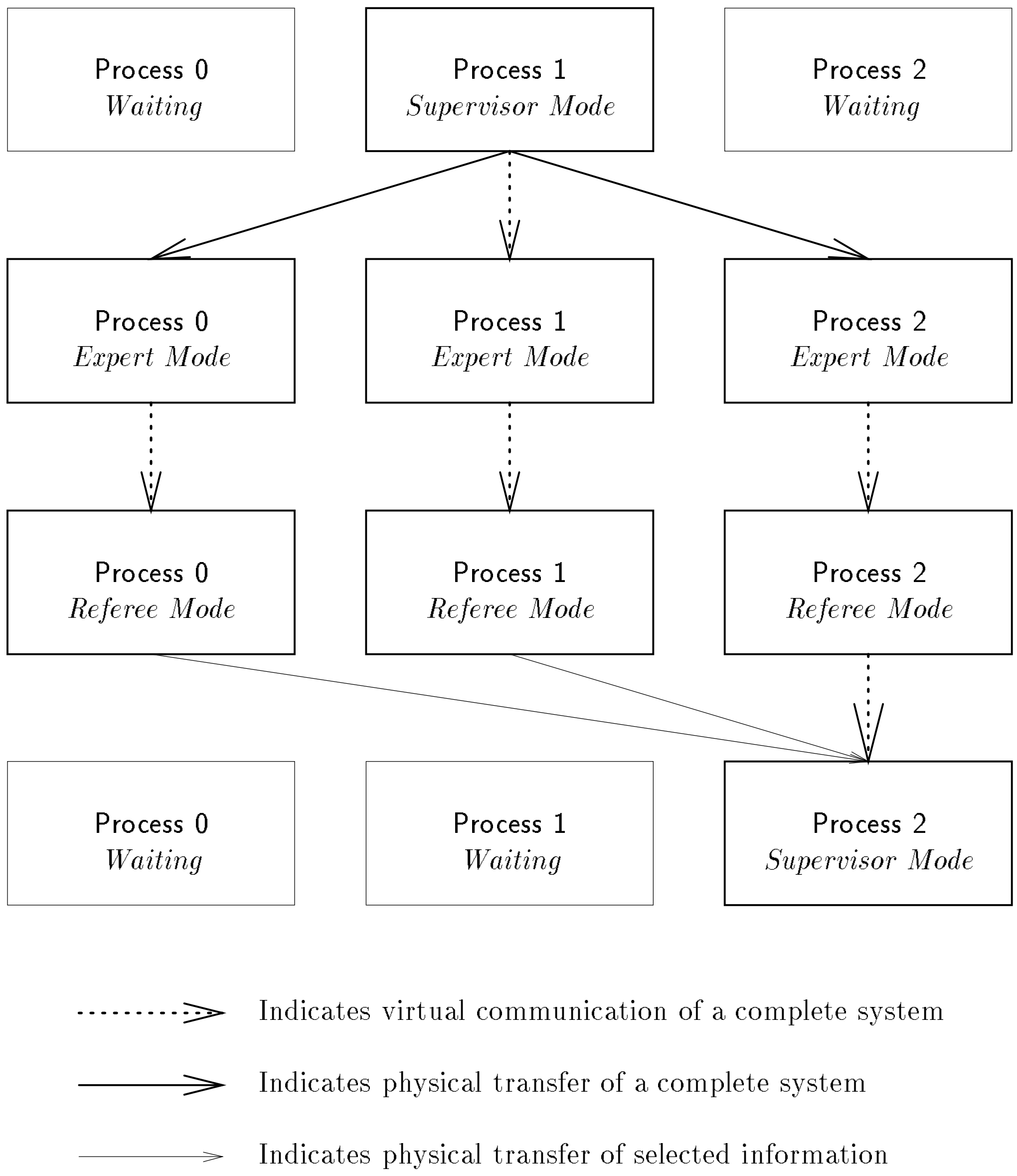


Figure 2.3: A cycle between two team meetings

internal representation of the proof is not feasible. This conclusion lead to the development of an external protocol language called PCL (*Proof Communication Language*).

PCL is a language with a very simple structure. A proof process is represented as a number of successive *steps*, labeled by unique identifiers. Each step contains one *fact* (a rule or equation) and an expression describing its origin. This expression contains the references to the steps used in the generation of the new fact and a complete description of all the inferences done on them.

Please note that a PCL listing describes a directed graph, with the steps at the nodes and the links going from a referenced step to the referencing one. This graph can be unfolded to form a *proof tree*.

In order to aid the analysis the steps also contain type information about their facts. Example types are *goal*, *lemma*, *rule* or *equation*. The current version of PCL is only aimed at completion based equational provers. However, the extension to any inference based reasoning process is straightforward.

A PCL listing is a complete, prover-independent description of the proof process. PCL is a purely sequential language. However, due to the well-defined communication in the TEAMWORK approach we were able to describe distributed proofs in PCL, also. Proof reproduction techniques enabled us to generate proof descriptions without disturbing the prover in any measurable way.

[DS94a] is a comprehensive guide to the latest version of PCL. It contains a complete description of the language, and also relates *en detail* our approach to the recording of TEAMWORK proofs.

In addition to the language we also developed a number of tools to aid in the analysis of PCL listings. The fist stage in the analysis is the *pruning* of unnecessary steps from the usually immense listing. Only after this process has condensed the data to a manageable amount, an in-depth analysis is feasible. Most of our manual work concentrates on the pruned listings. Section 4.1 presents some of the analysis results relevant to this work.

Further automatic processing introduces *lemmata* into the listing, marking especially important steps. Finally, the inference-based description of the proof is transformed into a calculus humans are used to. The resulting proofs in the form of equational chains can be read and understand by people without any background in automatic theorem proving.

Chapter 3

Knowledge Use in Distributed Equational Reasoning

As we defined in section 2.1, learning is a process of knowledge acquisition. If we want to use learning techniques to improve the performance of a distributed equational proof system, we first have to analyze where knowledge is used to guide the reasoning process. In this chapter we take a look at the different kinds of knowledge used to guide a TEAMWORK based proof system.

There are three broad levels of granularity in a TEAMWORK based equational prover – the *team* level, the *expert* or heuristics level and the *inference step* level. Figures 3.1 to 3.3 visualize these levels. Knowledge is employed in each of them, and consequently each of them offers some possibilities for learning mechanisms.

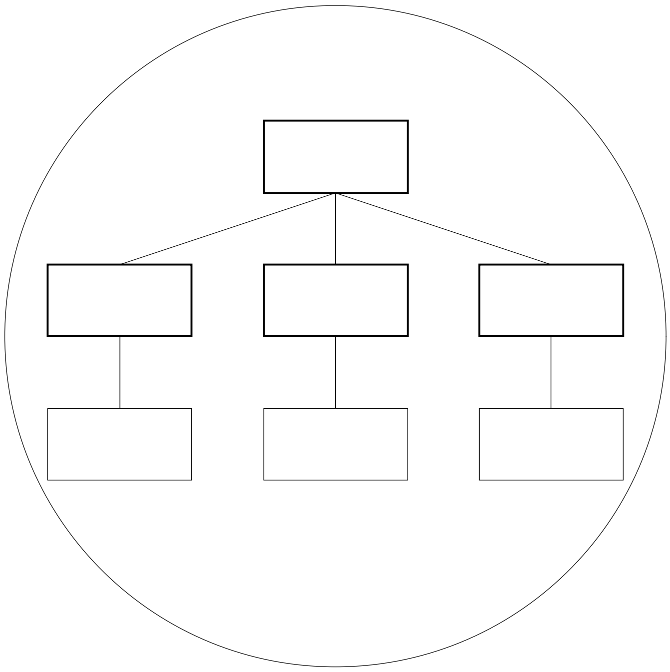
3.1 Knowledge at the Team Level

The supervisor has a large number of experts and referees for the composition of a team at his disposal. Additionally, a number of global parameters are guiding the behaviour of the team.

Planning knowledge is used to select a good combination of team members and global parameters. Included are information about good initial teams for specific domains, the referees suitable for a given expert, the cooperation among different experts, and adequate durations for the team cycles. This knowledge about the behaviour of the complete team resides with the supervisor. Additional knowledge about the evaluation of experts resides with the referees.

The effect of using planning knowledge is important especially for untrained users. The system can adapt itself to different problems. The user is relieved of the burdensome task of finding a good configuration. Thus, even inexperienced users can achieve interesting results with the prover. Up to now, planning knowledge for DISCOUNT is provided by experienced users in the form of domain description and general plans. Using learning techniques to acquire the relevant knowledge has two main advantages:

- By observing the behaviour of the team in a given domain, suitable configurations can



The supervisor has knowledge about the complete team,
the referees evaluate the experts with respect to the team.

Figure 3.1: Knowledge at the team level

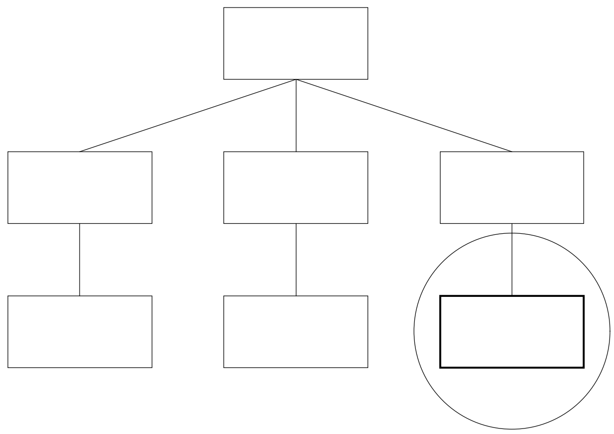
be recognized without the need for any user interaction. The adaption to new domains becomes much easier.

- As a computer can handle large amounts of data much better than most humans, it might even be possible to find correlations human operators miss. This can allow for a more optimal adaption of the system.

There are, however, a couple of problems associated with learning at the team level.

- The raw power of the prover is not significantly affected. Only the user friendliness is improved.
- At least for long term learning (as opposed to situational adaption) each team run yields only one example. Thus, only a small number of examples is available.
- Only long and complex proof runs yield interesting data. For simple examples no intervention of the supervisor or the referees is necessary, and thus no knowledge about the complete team is gained.

Improvements of the supervisor have recently been integrated into the DISCOUNT system. The supervisor is now using a user-programmed *long term memory* to compose good teams



Knowledge of the experts covers only aspects of completion.

Figure 3.2: Knowledge at the expert level

for known domains. It is also monitoring the progress of the team and uses *reactive planning* to change the team to adapt to a given problem.

This approach has been quite successful. [DK94] describes the work in more detail. However, automatic knowledge acquisition at the team level is neither included nor planned for the near future.

3.2 Knowledge at the Expert Level

Each of the experts in the team basically implements the unfailing completion algorithm from section 2.2.2. This algorithm contains two places where indeterminism has to be eliminated by the use of knowledge: The *reduction* strategy used for normal form computation and the strategy for the *evaluation of critical pairs* to be inserted into the set of unprocessed critical pairs.

The choice of a general reduction strategy happens on the expert level, while single decisions for each possible reduction would belong to the inference step level. For simplicity, both are treated in this section. Improving the reduction strategy can be advantageous in certain cases:

- It is well-known that a clever reduction strategy can lead to much shorter rewriting chains. For example, the bound for the length of the simplification chain can change from an arbitrary recursive function to constant or linear if the reduction strategy changes from innermost to outermost or vice versa.
- For most of the proof process the rewriting relation defined by the rules and equations is not confluent. Thus, selecting the best of many normal forms might speed up the completion.

These advantages are offset by a number of disadvantages:

- Long reduction chains do seldom appear in real proof problems. Even the overall time spend for reduction in the DISCOUNT system is rather small and could be further reduced by using techniques like *indexing* for the rules and equations, thus speeding up the reduction without the need for an extensive knowledge base.
- Although they only take up a small fraction of the run time, reductions are attempted very often in a completion based prover. Any overhead added to a reduction would significantly affect overall efficiency.
- Most rewrite systems at the moment use only one reduction strategy for a given problem. Thus, getting examples using different reduction strategies would be rather difficult.

Changes in the critical pair evaluation have up to now been the main reason for improvements in completion based provers. However, the analysis of proofs has shown that even in proofs generated by very strong teams there still is immense room for improvement. The next chapter will deal with these topics in detail. Learning good evaluation strategies for critical pairs has a number of promising properties:

- The basic completion algorithm is unaffected. Changes to existing systems can be kept very local.
- For known problems and sub-problems an optimal critical pair selection strategy can be learned.
- Each proof uses a large number of equations. Proof protocols show these equations and their importance to the goal. Thus, an abundance of examples exists.
- First analyses of proofs show that certain equations appear in very different proofs. A certain extrapolative power to unknown problems can be expected.
- Various techniques can be used to recognize important results not only on a global scale but also for more specialized situations. Proofs for certain goals or in special domains are examples.
- For the DISCOUNT system the implementation effort can be reduced by reusing some routines for domain recognition already implemented to support the supervisors planning work.

However, a number of minor problems and disadvantages can also be identified:

- The number of critical pairs generated during an average proof run is extremely high. In order to handle this amount of data, some restrictions are necessary. Learning from negative examples (useless equations) seems to be impractical.
- Even for an optimal critical pair selection unnecessary steps cannot be avoided. Each critical pair will trigger a large number of operations, only a few of which need to further the proof.

- If the critical pair selection is the only feature of a proof relevant to the learning process, the prover may actually learn suboptimal (but working) proofs. This problem is shared by most learning problem solvers, though.

Learning of the critical pair selection strategy will be the main focus of the following chapters. As TEAMWORK's experts are only distinguished by their evaluation strategy, we will use the terms *learning strategy* and *learning expert* synonymously to denote an evaluation strategy that improves with knowledge added in the form of example proofs.

For a TEAMWORK based system, learning at the expert level has one additional advantage: The team can always be expanded to include not only the learning experts but also a healthy collection of conventional experts. This minimizes the risk that the overhead necessary for the learning experts can harm the overall performance.

3.3 Knowledge at the Inference Step Level

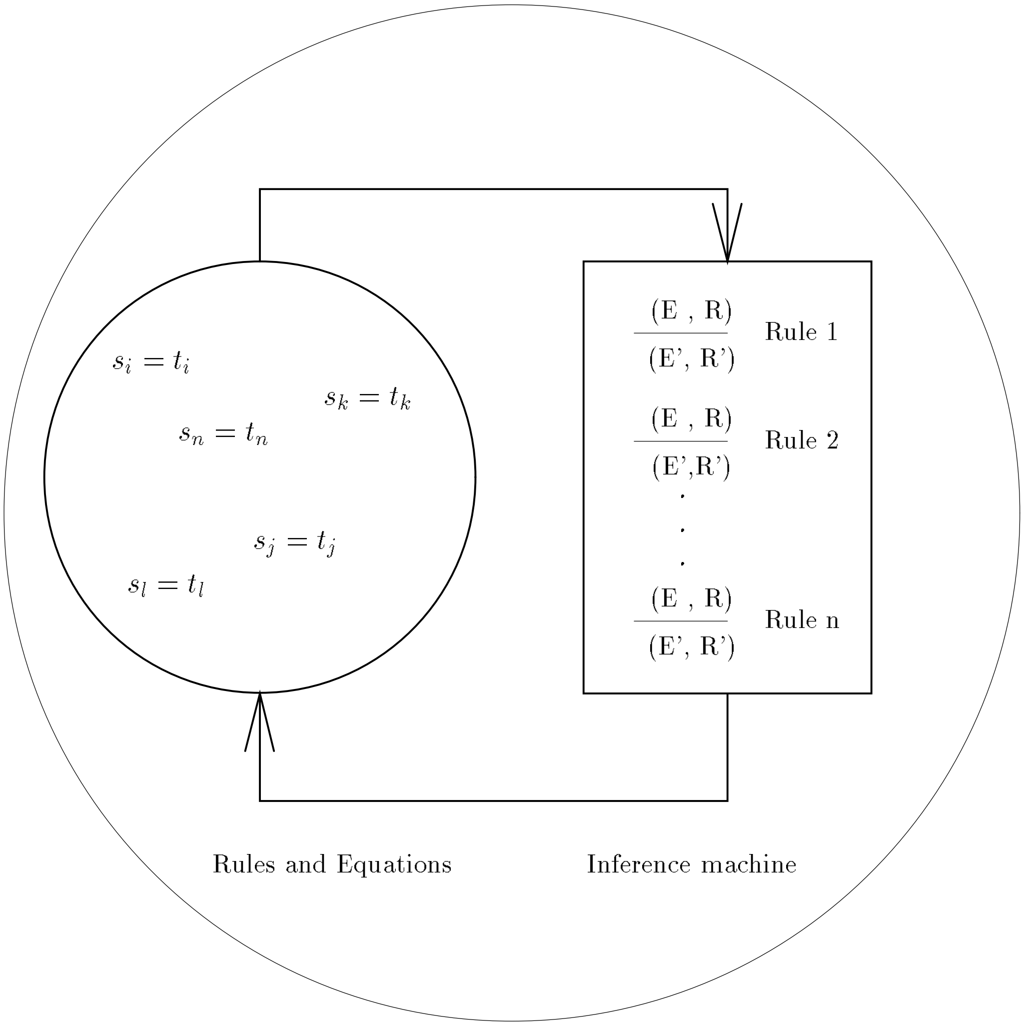
The algorithm presented on page 21 and used as the basic element in the DISCOUNT system is only one possible implementation of a prover based on the inference rules from section 2.2.2. It is possible to implement a prover who does work in a much less rigid way, controlling every single inference done. Such a prover could learn to perform the most promising inference at each point in the proof process. This approach offers a variety of interesting features:

- The reasoning and learning paradigm seems to more closely resemble the way humans reason about equality. Only promising conclusions are drawn.
- Proofs can be performed in an optimal way, without any overhead created by unnecessary inferences. A learning prover can, in theory, learn to avoid *all* unnecessary inferences.

However, there are two grave disadvantages that make an actual implementation less than promising at the moment.

- The implicit control knowledge provided by the completion algorithm has to be made explicit. In order to ensure completeness, this knowledge has to at least include the status of each *potential* critical pair. The resulting overhead would seriously impair the power of any system.
- Recognition of analogue proof situations would be a very expensive operation, as a match against the complete state of the proof system would be necessary. Even if only isolated parts of the state are used for the match, a very large search space has to be covered for each step.

Thus, while learning at the inference step level is a very interesting concept, the costs introduced by technical problems are probably too high for a practical application.



Knowledge at the inference step level guides the application of single inference rules

Figure 3.3: Knowledge at the inference step level

Chapter 4

Machine Learning for Unfailing Completion Heuristics

A sufficiently powerful inference machine is a prerequisite for a powerful proof system. However, our experiences have shown that good proof strategies are even more important - especially for non-trivial examples.

The most naive strategy is the "first in – first out" (*fifo*) approach. Critical pairs are selected for processing in the very order they have been generated in. This strategy is obviously fair, however, its performance is very poor. More elaborate conventional strategies use syntactical measures on the critical pairs. Very good all-purpose heuristics prefer equations where either the sum or the maximum of the term weights (a weighted sum over the variables and operators in the terms) is small.

Some criticism has been directed at completion, because these conventional strategies do not take the theorem to be proved into account. As a result, new, goal oriented heuristics have been developed. Anantharaman and Andrianarievelo in [AA90] used structural *measures* on the goal and the equations to eliminate critical pairs unlikely to contribute to the proof. This approach evolved into the *occnest* expert for DISCOUNT (see [DF94]), which uses the measures not to totally exclude critical pairs, but to establish a preference ordering among them.

Independently, similarity-based heuristics have been developed by the DISCOUNT team (also presented in [DF94]). Matching and unification between subterms of critical pairs and the goal are used to evaluate the critical pairs. These heuristics are very successful in speeding up the final phases of the proof.

The combination of various heuristics in the TEAMWORK approach further contributed to an increased power and user friendliness of the proof system. Our proof analysis has revealed not only the degree of the improvements for good strategies, but also some of the reasons for these improvements.

Problem	Complete	Pruned	Comments
Lusk2	54	16	Group axioms imply $(x^{-1})^{-1} = x$. This is a very simple problem from [LO82].
SelfInverse	1703	69	In a ring with $x^2 = x$ every element is self inverse, that is $x^{-1} = x$. This is related to the next problem.
Lusk3	5009	83	In a ring with $x^2 = x$ the multiplicative operation is Abelian. See [LO82] for a discussion.
Cooperation	98532	64	See table 4.2.
DeMorgan	238706	151	ditto
Luka1	145078	22	ditto
Luka2	85934	58	ditto
Luka3	322001	79	ditto
Lusk4	13420	95	In a group with $x^3 = e$ the equation $h(h(x,y),y) = e$ holds for the commutator h. See [LO82].
Lusk5	46477	45	See table 4.2.
Lusk6	387273	190	ditto
p03a	3146	25	ditto
lat4	485010	139	ditto

Table 4.1: Numbers of steps in complete and pruned PCL listings of sequential proofs

4.1 Results of the Proof Analysis

4.1.1 Performance of Search Strategies

The PCL protocols of proof runs reveal a number of interesting features. First, proofs for most problems are rather small. Only a couple of inference steps are needed – most pruned proof protocols contain only a few hundred steps at the absolute maximum. Even very difficult proofs rarely need significantly more than 200 proof steps. However, to arrive at the proof the prover has to traverse an enormous search space. Complete listings containing tens or even hundreds of thousands of inferences are not unusual, even if powerful heuristics are used to speed up the search. Table 4.1 shows the numbers of inferences done and the number of inferences actually contributing to the proof for a couple of examples[1]. Please note that the table contains data for the best (fastest) sequential proof for each example. Only the first, trivial example in the table can be proved with the *fifo* strategy in any reasonable time.

This data obviously shows a huge potential for improvements – only a miniscule fraction of the proof steps is actually useful. TEAMWORK, in addition to bringing more processing power to bear on the problem, realizes a certain amount of this potential. Table 4.2 contains the results for proofs found by teams. For problems present only in table 4.2 DISCOUNT does not find a proof using any conventional strategy in sequential mode[2]. Problems only

[1] A more detailed discussion of these experiments and exact specifications of the examples can be found in [DS94a].

[2] For BoolAssoc, as an example, the sequential prover did 879594 inferences (generating a protocol of 73 Megabytes) before terminating unsuccessfully due to lack of memory.

Example	Complete	Pruned	Comment
BoolAssoc	95788	117	The disjunctive operator (`and`) in a Boolean algebra is associative.
Cooperation	6879	58	Calculations in a cyclic group. Presented in [Pi92].
DeMorgan	181336	159	Proof of one of DeMorgan's laws in an arbitrary boolean algebra.
GT7-3	9638	214	Derivation of the associativity axiom from a single equation axiomatization of a group. See [LW92].
Luka1	139178	22	Derive the first of Lucaciewicz's axioms for propositional calculus from Frege's axiomatization of this calculus. See [Ta56].
Luka2	47898	57	Derive the second of Lucaciewicz's axioms.
Luka3	180278	42	Derive the third of Lucaciewicz's axioms.
Lusk5	24738	48	An example using a ternary Boolean algebra – see [LO82].
Lusk6	144099	227	In a ring with $x^3 = x$ the multiplicative operation is Abelian. This was presented as a challenging example in [LO82].
p03a[3]	9500	18	In a lattice ordered group the group operator has certain monotony properties (a conditional conclusion).
p10	28124	103	In a lattice ordered group the inverse of a least upper bound of two elements is the greatest lower bound of the inverse elements.
lat4	63860	133	Show that each element of a lattice ordered group can be expressed as the product of its positive and its negative part.

Table 4.2: Numbers of steps in complete and pruned PCL listings of distributed proofs

present in table 4.1 are proved by the distributed system before the first team meeting takes place. In this cases no real distributed proof is generated.

Comparing the two tables shows that, despite the fact that more than one expert works at the problem, the overall number of proof steps done by the distributed system is lower. Thus, the combined strategy of the team is actually an improvement over even the best sequential strategy for the problem. Competition and cooperation among the team members both contribute to this effect. See [DS94b] or [DS94a] for a more detailed description of the phenomenon.

The results presented show that TEAMWORK does significantly reduce the actual search space traversed. However, it is just as obvious that the overhead due to imperfect strategies is still immense. There still is a lot of room for improvements in the search strategy.

[3]For this example the sequential expert performed actually better. The *occnest* expert is extremely well adapted to the problem, and the team only results in overhead. It should be noted that no other conventional heuristic could prove this example at all, and that a team without occnest needs more than 320000 steps to find a proof.

Processed pairs	**Lusk3**			**p03a**			**BoolAssoc**		
	fifo	*add*	*occnest*	*fifo*	*add*	*occnest*	*fifo*	*add*	*occnest*
0	9	9	9	16	16	16	7	7	7
5	9	5	6	21	11	11	23	5	5
10	23	12	29	28	10	27	27	1	55
20	135	36	102	115	10	39	361	29	177
50	548	214	365	519	89	291	1768	158	151
100	2470	475	1077	2088	680	–	100	620	285
200	8881	719	–	4127	1401	–	14370	2565	522
300	30176	–	–	7185	2362	–	> 36024	4033	4167

Note: A dash (–) as an entry indicates that the prover did not reach this number of processed pairs because it terminated successfully. The *fifo*-strategy did only manage to process 287 critical pairs for the BoolAssoc example in 12 hours on a SPARCStation 20.

Table 4.3: Branching factors for three examples and different strategies

4.1.2 Search Space Control and Goal Orientation

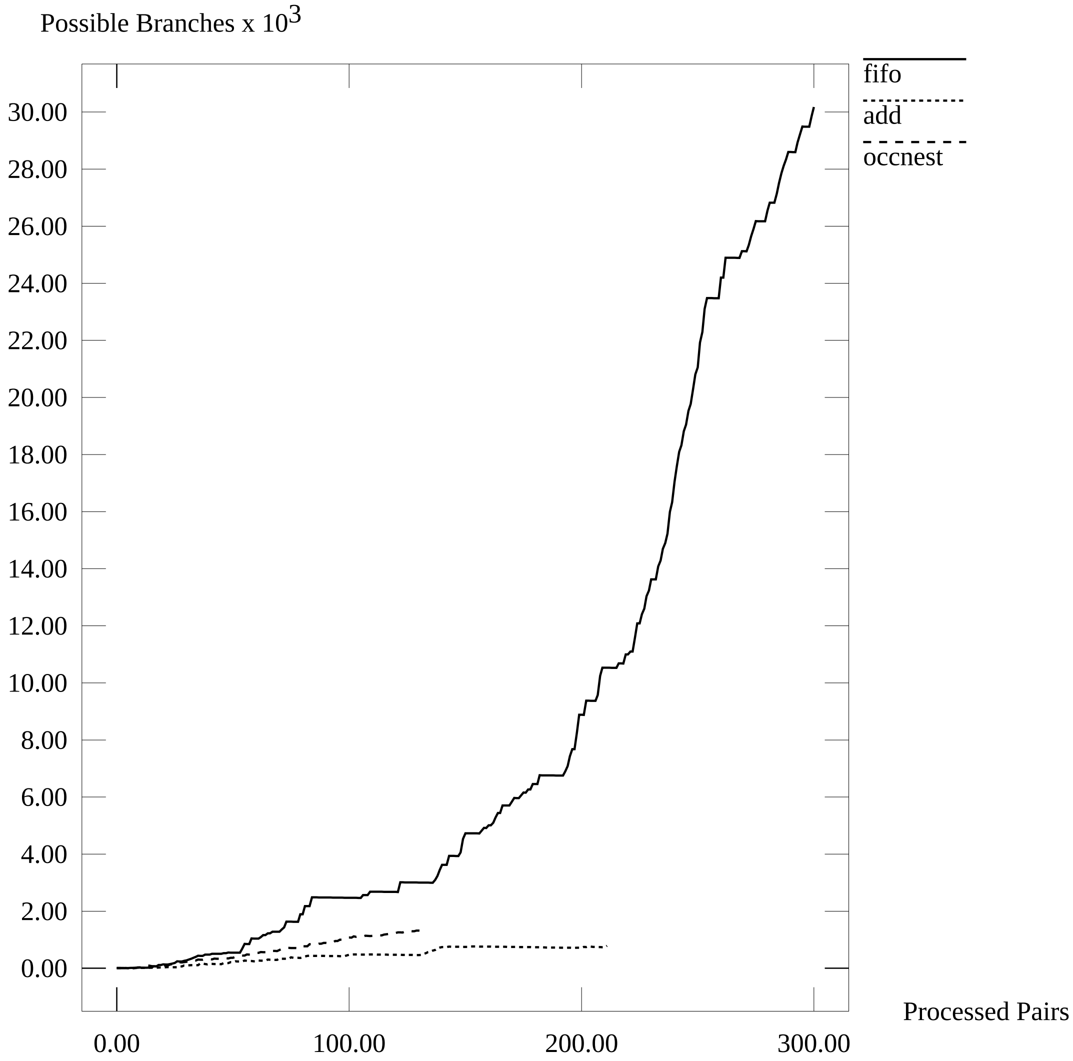
Table 4.3 shows just why the control problem for completion is so difficult. It contains the branching factors involved (critical pairs ready to be processed) versus the number of processed critical pairs for three problems from different domains and three typical strategies. The tests were run up to 300 processed critical pairs. Figures 4.1 to 4.3 show the complete data for the three examples in a graphical presentation.

The *fifo* strategy was mentioned above. It simply processes critical pairs in the order they are generated in. The *add* strategy is a prime example for pure completion strategies. It prefers critical pairs consisting of small terms. *occnest* was also briefly mentioned above. It is a new, goal-oriented strategy developed especially for equational theorem proving (as opposed to pure completion). The *occnest* strategy is very sucessful, particularly in the domain of *lattice ordered groups* that will be used as a test case throughout this work. At the moment it is unknown whether *occnest* is fair. Opinions differ, and a proof has not yet been proposed.

The figures show exactly how the different strategies perform. The *fifo* strategy is the least successful. There is very little intelligence or knowledge in the evaluation of the critical pairs. Consequently, the search space explodes very early and in no case the proof was found within the 300 pairs limit. For the example BoolAssoc the prover did not manage to process 300 critical pairs in a reasonable time (12 hours) on a SPARCStation 20. In this case extremely large rules are generated very early. Rules were almost never simplified – after processing 200 critical pairs 59 out of a total of 61 generated rules were still part of the system. These factors contribute to an extremely unusual load on the inference machine.

The *add* strategy performs very differently. It took less than 20 seconds to process all 300 critical pairs on the SPARCStation 20. Due to a very strong selection criterion the search space expands much more slowly. A strong system of rules and equations is generated early and allows the prover to cut down on redundant data, thus limiting the search space further.

The *occnest* strategy performs, on average, slightly worse than *add* with respect to the search space explosion. It is much better than *fifo*, though, and is able to select critical pairs

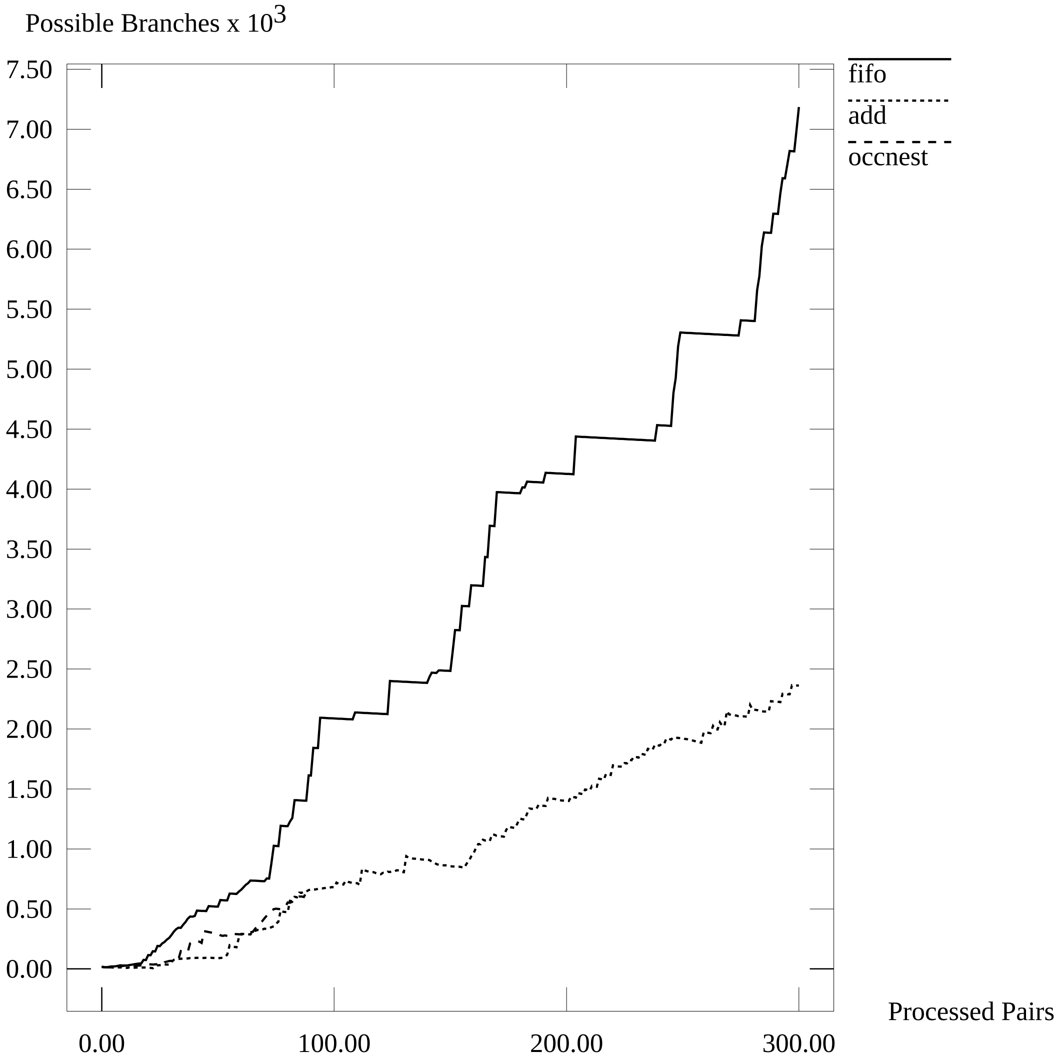


Note: For the strategies *add* and *occnest* the system terminated after processing 211 and 139 critical pairs, respectively.

Figure 4.1: Branching factors for Lusk3

particularly useful in proving the given theorem.

Thus, we can identify two explanations for the success of a strategy. First, selecting critical pairs that lead to a strong knowledge base and manage to restrict the search space explosion are very likely to be successful in many cases, independent of the theorem. In a TEAMWORK-based prover, strategies utilizing this approach are particularly useful in early and medium phases of the proof, because they establish a strong knowledge base early on. Secondly, finding critical pairs particularly tailored to prove a specific goal is another useful tactic. Strategies working along these lines are probably useful in the later stages of the proof.



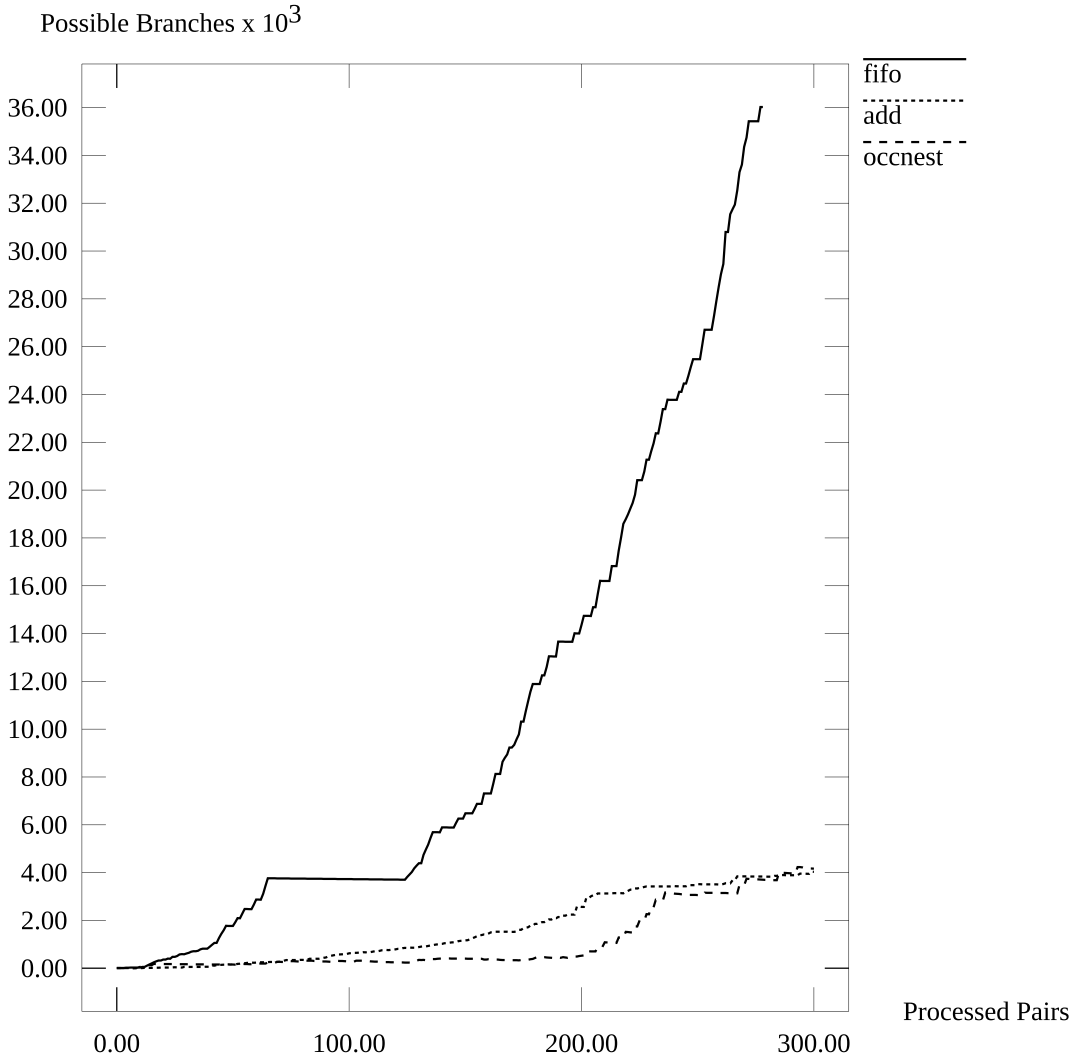
Note: Using *occnest*, the prover terminated successfully after 89 processed pairs.

Figure 4.2: Branching factors for p03a

4.1.3 Other Observations on Proofs and Proof Processes

Some additional observation about DISCOUNT proof processes should be noted:

- We have been working with examples from a number of different domains – rings, lattice ordered groups, and propositional calculus are three important fields. In each of these domains we encountered many important lemmas. These were used in most proofs. Some lemmas even occured in more than one domain. This indicates that equations with certain structures may play important roles in a wide variety of proofs.



Note: For the *fifo*-strategy the prover was terminated prematurely after 12 hours. It processed only 278 critical pairs in this time.

Figure 4.3: Branching factors for BoolAssoc

- The referees implemented in the DISCOUNT system use only very simple statistical criteria for their evaluations. However, the proof analysis showed that they already perform surprisingly well. Steps selected by the referees as outstanding have a very high probability of furthering the proof process - on average, about 30 % of these steps are useful, while only 0.2 % of all steps are useful. Of course there is still room for improvements. However, the problem does not seem to be a particularly pressing one.

- Only a limited number of experts can be employed usefully in a given proof. For most proofs this number is two or three. Additional experts may raise the chance that a good combination of provers is found. However, only two or three experts will contribute to the proof. We believe this to be due to the fact that current examples only require two or three main ideas to be followed. Whether this trend will change if more complex problems become common is, at the moment, unknown.

4.2 Related Work

Using learning techniques to improve the behaviour of problem solvers is no new idea. However, most techniques so far have been restricted to systems that use backward reasoning employing top-down decomposition techniques. We are not aware of any approach for basically forward-oriented completion based equational theorem provers.

The STRIPS system ([FN71]) was one of the first problem solver to use learning techniques. STRIPS was a system for the generation of robot plans. It worked by applying a set of operators to world models, thus reducing the difference between an existing and a desired state. Composite goals would be split into subgoals and handled one after the other using a stack. An engine to learn from previous plans was added to the system ([FHN72]), improving its behaviour significantly. Old plans were analyzed, generalized as much as possible (while maintaining correctness), and stored. For later planning attempts these plans would be used as macro operators. Partial plans could be extracted and applied in appropriate situations. While STRIPS predates the concept of explanation-based learning, most of the ingredients are already incorporated into this system.

In a similar approach, the SOAR problem solver for industrial applications was extended using explanation-based generalization (compare [RL86]). In a first approach parts of solutions are converted into macro-operators. The authors coined the term *chunking* for this operation. A later enhancement added a full explanation-based generalization engine and the possibility to restrict the sometimes overly general rules to narrower cases.

Probably one of the most successful and best known learning system is the problem solver PRODIGY ([CE+91]). PRODIGY employs a number of different approaches for learning: Veloso and Carbonell ([VC91]) integrated *analogical replay* as a case based learning mechanism related to Carbonell's *derivational analogy*. In this approach previous plans are accumulated. They are generalized only if two analogous problems can be merged into a single plan (as opposed to the eager generalization in EBG approaches). The authors report a 20 times improvement in performance and stress the importance of a dynamic and steady transition from pure replay to case adaption to from-scratch proving – a point also emphasized by many other authors.

PRODIGY also uses explanation-based learning for the acquisition of *search control knowledge*. Rules of the form *if(test(state)) then do_action(state)* are used for distinguishing among appropriate and inappropriate choices during the search. These rules are generated using "partial and plausible explanations in an imperfect world" ([MC+89]). The EBL engine has been the subject of some work. In [MC+87] and [Mi88] the authors deal with the cost/benefit relation of EBL. They devise a mechanism to filter out unprofitable learned rules, thus reducing the cost to find rules matching the problem state.

Etzioni in [Et90] and [Et93] makes another point. His analysis of PRODIGY's performance in differently specified, but equivalent domains led to the observation that EBL has problems with recursive proofs, i.e. proofs where the completion of a task is broken down into a number of similar, but smaller tasks (a typical example from PRODIGY's blocks world domain is the moving of a stack of n boxes by moving one block and moving a stack of $n-1$ boxes). Rules learned from non-recursive proofs did significantly help in the reasoning process, while rules corresponding to recursive proofs might even be an obstacle instead of an asset.

The PRODIGY system also features a mechanism for the automatic generation of different abstraction levels (see [Kn94]). This mechanism generates an abstraction hierarchy for a given problem, the system then solves the problem on a high level of abstraction. The abstract plan is then refined through the levels of the hierarchy until the desired solution is matched. The EBL engine and the abstraction mechanism have been integrated, and significant synergy effects have been observed ([KME91]).

[Bh90] offers yet another approach. In this work EBL mechanisms are used to explain (and learn) the dead-ends in the search space. *Heuristic censors* are implemented to prune potentially useless search paths. As the system allows for (incorrect) over-generalizations, some recovery mechanisms are suggested to maintain completeness.

In an area related to completion based provers, Larid and Gamble ([LG90]) extended EBG to speed up the evaluation of functional and declarative languages implemented by rewriting systems. However, no connection to completion and proof procedures is made – the authors are only concerned with the rewriting process.

In [KW94] Kolbe and Walther suggest explanation-based generalization to guide inductive equational reasoning. For inductive reasoning the main difficulty consists in finding good inductive lemmas to aid the proof process. A proof is generalized and compressed into a *proof catch* containing only the relevant data for the proof: The theorem, the axioms and eventual lemmas. If later instances of the same problem are encountered, the proof catch is instantiated and used to directly solve the new problem. The PLAGIATOR system implements these ideas. However, it is mainly targeted at proof reuse and does only possess very limited reasoning capabilities. The human operator is expected to provide most of the original proofs. Final results to the viability of this approach are not yet available.

4.3 Implications for the Learning Paradigm

Both the analysis of successful completion strategies and the experiences with learning problem solvers from other areas yield valuable data. They suggest a number of points to be kept in mind while choosing a learning paradigm for completion based provers.

- A good proof strategy might work by prefering critical pairs that have a high likelihood of improving the quality of the provers knowledge representation by generating strong, useful rules for the completion, thus cutting down on the search space explosion. It should be able to select critical pairs particularly likely to be useful with respect to the actual theorem at hand. Of course a strategy that combines these approaches might be particularly powerful.
- Full integration of proof replay, proof adaption and from-scratch proving is a very desirable feature. To allow this a strategy making use of learned knowledge has to interface to the prover in the same way a conventional strategy does. For DISCOUNT this implies that the learned knowledge has to be used in the evaluation of critical pairs.
- The learner should be able to degrade gracefully if it has no or little knowledge about its field. On the other hand, adding more knowledge should not (or at least not signif-

icantly) degrade the performance in other regions.

- Efficiency is an important topic. The evaluation of a single critical pair has to be comparatively cheap for the strategy to be competitive. Typically, a strategy has to evaluate a much larger number of critical pairs than actually used in the proof. This might even necessitate the use of a sub-optimal, but cheap (in terms of cpu time and memory) solution instead of a more optimal, but expensive solution.
- For the same reason, the amount of work done during the proof run should be minimized. As much work as possible should be done beforehand, using systems independent of the main prover.
- Recursive proofs should be avoided in the learning process. For completion based proofs this is very much a hypothetical problem, as these proofs are basically forward oriented and use no recursion at all.
- Learning from negative examples – either failed proofs or unused equations in successful proofs – does not seem feasible. The enormous amount of data available would make learning extremely slow and tedious.

4.4 An EBL Approach to Guide Equational Deduction

The above observations suggest a learning concept that can best be described as explanation based. It is depicted in Fig 4.4.

The knowledge acquisition phase is very similar to strict explanation based generalization procedures. First, a proof protocol in PCL is generated. This is most probably done by the learning proof system itself, though other sources, like user-interactive proof tools, are imaginable. This proof protocol is pruned using the existing tools, yielding a proof tree devoid of all unnecessary information.

The facts in this proof tree are then generalized and reorganized. The relevant information about their role in the proof (implicitly given by the structure of the tree) is made explicit, and the structure of the proof tree is discarded. This step differs from strict EBG. First, as the learned knowledge is only used in evaluating single critical pairs, the structure of the proof can be completely discarded once the relevant data has been extracted.

Secondly, we do not use the equational domain theory to ensure that the generalizations are valid, but generalize using the incomplete theory that the structure of the terms are a valid indicator for the critical pair's quality. As we use the knowledge only in evaluating new critical pairs, and as the inference mechanism of the prover ensures that all critical pairs are indeed valid in the current context, no risk is incurred.

The knowledge generated in this way is not stored isolated for each proof, but organized in cases subsuming several proofs sharing significant features. We use three different methods to build cases: Cases identified by goal, identified by specification, and one global case containing information from *all* proofs. For each of these methods, different degrees of generalization are performed on the new knowledge. It is then merged into the existing knowledge base. The prover can access this knowledge base and use the stored knowledge in a variety of ways to favorably guide the reasoning process by selecting good critical pairs.

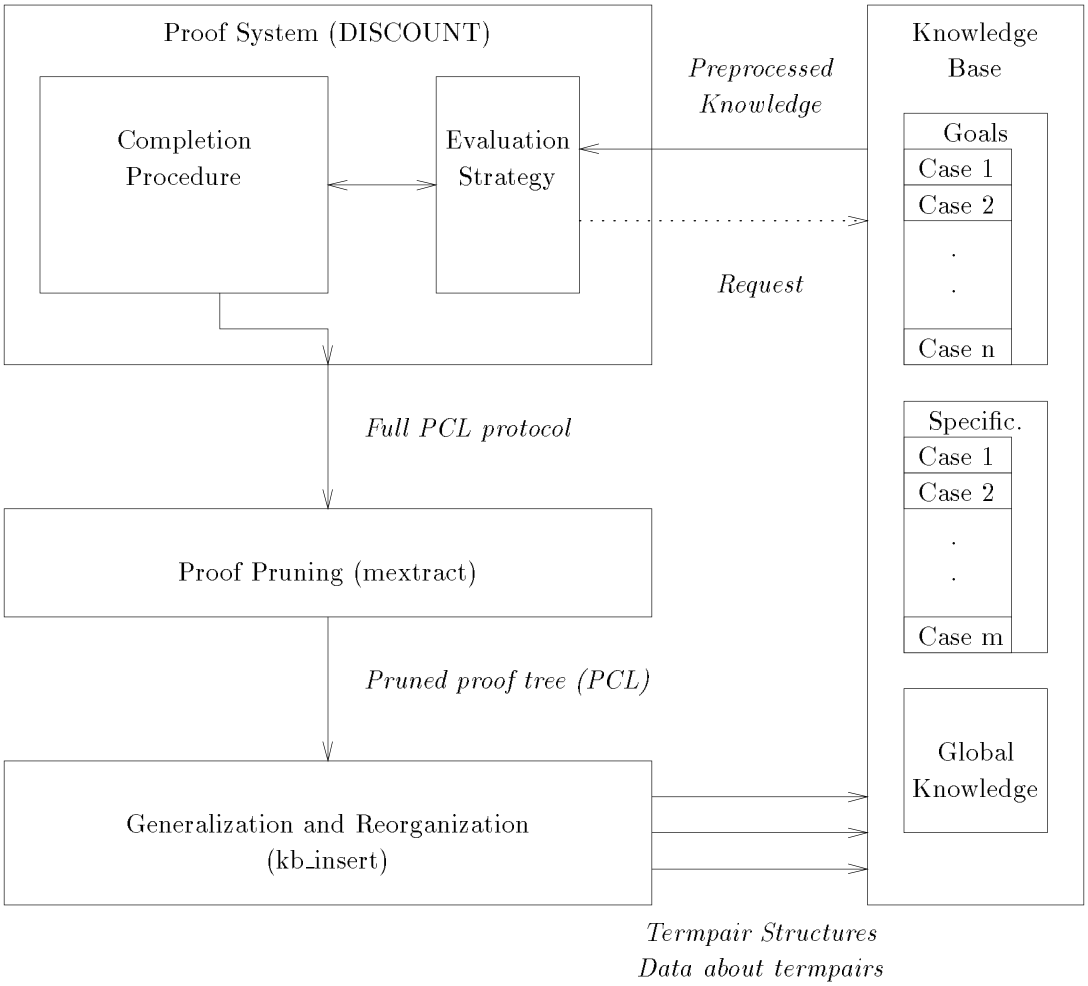


Figure 4.4: Learning for equational deduction

For a TEAMWORK prover, each of the experts in the team implements a full completion procedure. The proof is delivered by the complete system, but each single member may access the knowledge base.

This concept of learning from protocols of successful proofs can handle the full spectrum from entirely closed-loop learning (only proofs generated by the learning system are fed back into it) to entirely open loop learning (only proofs found by other means are given to the learning component). In most cases a mixed operation will be desired and is perfectly supported.

Chapter 5

The Knowledge Base

This chapter discusses how the knowledge for the learning experts can be stored in a way so that it can be accessed and used efficiently. The observations from the previous chapter already suggest some hypotheses:

- The knowledge representation should be pre-processed to allow the prover to access it in a more efficient way.
- Only knowledge gained from successful proofs will be considered to avoid being overwhelmed by the data.
- The access to specific parts of the data should be very efficient. The absence of requested information should also be cheap to confirm in terms of cpu time.
- As memory shortage always is a problem for completion based provers, the knowledge should be stored in a very compact way.
- The adaption to similar cases should be a simple operation. In particular, the knowledge should be stored in a form independent of trivial details like the names of operators or variables.
- If possible, existing information and data structures within the DISCOUNT system should be reused to avoid any overhead.

We want to store all the equations responsible for the success of a proof. As we want the learning expert to be able to extrapolate beyond the scope of the stored proofs, we do need to gain general knowledge from the concrete proof. As stated in the previous section, this knowledge will be organized in three different ways – in cases (or *domains*) defined by a common *goal* or theorem to be proved, in cases defined by the common *specification*, that is, the axiomatization of the problem (taking only the axioms that were actually needed into account), and finally as global knowledge not indexed by any characteristic elements of the problem.

For all three approaches we need a way to generalize the relevant equations. For the goal and specification bound knowledge we also need to identify the case the knowledge belongs to. Finally, for the reuse of knowledge we also need a way to efficiently find the knowledge pertaining to a particular equation in the global knowledge base or one of the cases.

A customary solution for the indexing of data about proofs is to generalize the clauses by substituting the function symbols with arbitrary second order variables. To reuse the proof, the key characteristics of a case are instantiated with all possible signature matches, one at a time, and compared to the problem at hand. If the key elements are recognized, the current match is selected to re-instantiate the complete case. Unfortunately, the number of possible signature matches grows exponentially with the size of the signature, and evaluating the match for suitability can be quite expensive. Nevertheless, we (partially) use this approach for the specification bound information stored in the knowledge base. However, there are less expensive ways of handling the other cases.

The choice of the retrieval mechanism is simpler. Binary search trees offer, on average, very good performance at low memory overhead. As the data base is generated independently of the proof run, we can even afford to organize the knowledge in balanced binary trees (AVL trees – see [Kn73, pp. 541–]) without burdening the prover. Balanced search trees guarantee very good performance even in the worst case. An additional advantage is that a previously computed AVL tree can be stored in a way that it can be rebuilt with a simple binary insert, without the need to check for balancing operations. However, in order to use search trees we need a total ordering on the possible keys.

The next sections will explain in just how the different parts of the knowledge can be handled. However, we first want to take a look at the global structure of a complete knowledge base.

5.1 Structure of the Knowledge Base

To increase the maintainability, the knowledge base is structured hierarchically. On the top level, the global knowledge and administrative data is stored. On the second level of the hierarchy, three different clusters contain the goal bound knowledge, the specification bound knowledge and, for the users convenience, the original pruned PCL listings inserted into the knowledge base.

The hierarchy is realized by four UNIX file system directories and a varying number of files. More exactly, the structure is as follows (compare Fig. 5.1):

- The complete knowledge base corresponds to a directory in the UNIX file-system. The name of the knowledge base is the name of this directory.
- The top level directory of the hierarchy contains four files and three subdirectories: The files `goaldoms`, `specdoms`, `pdata` and `cdata`, and the directories `EXAMPLES`, `GOALDOMS` and `SPECDOMS`.
- `goaldoms` contains the description of the different cases identified by a common goal. Only the information necessary to identify a case are stored in this file, the complete data for a case is stored in a separate file.
- `specdoms` serves the same purpose for specification bound knowledge. The file contains only the identifying data for the cases identified by a common specification.
- `pdata` and `cdata` represent the global, unbound knowledge stored in the knowledge base. `pdata` contains all the lemmas that were ever useful in a proof, while `cdata` contains information on each and every equation needed in a proof inserted into the base.

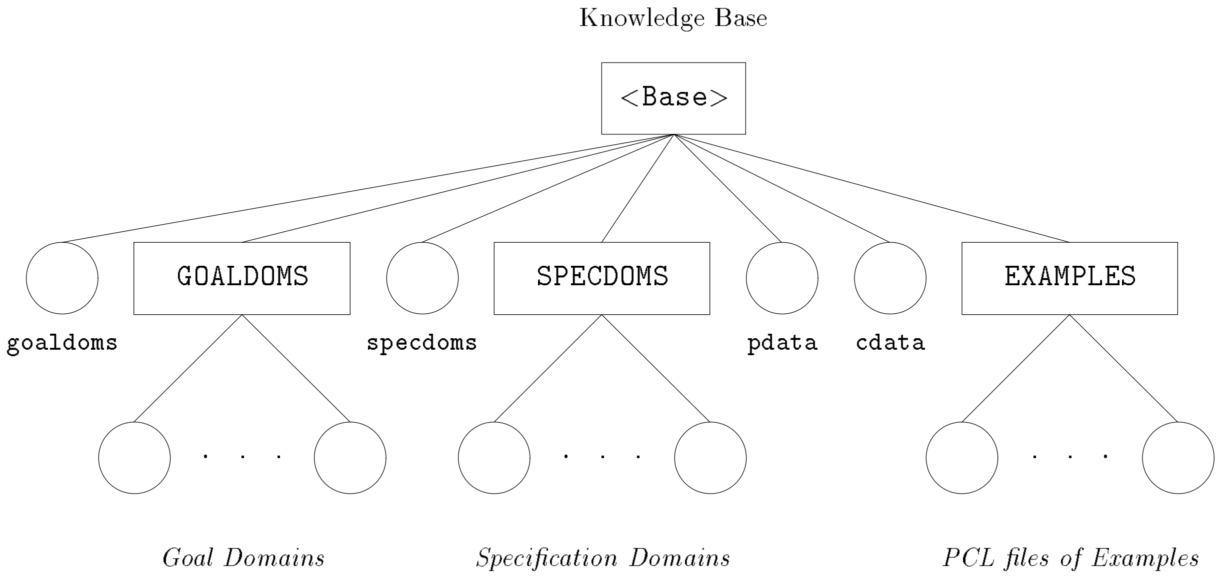


Figure 5.1: Structure of a knowledge base for DISCOUNT

- The directory EXAMPLES is used to store verbatim copies of each pruned PCL protocol ever entered into the data base. They serve mainly for control purposes and allow an easy merging of two knowledge bases by simply inserting the PCL protocols from one into the other.
- The directories GOALDOMS and SPECDOMS, finally, contain the different cases – one file for each goal domain and one file for each specification domain.

5.2 Data Structures for Terms and Equations

In this section we will introduce the representation for terms and equations used in the knowledge base for the DISCOUNT system. The basic idea is to substitute the function symbols in the term, in the order of appearance, with operators from $\{f_1, f_2, f_3, \ldots\}$ and to substitute the variable symbols with symbols from $\{x_1, x_2, x_3, \ldots\}$. This allows both the definition of a efficiently testable, total ordering on the terms and a simple, syntactical check to identify terms with identical structures. The following definitions cope with these aspects.

Definition 3 : Signature substitutions, Patterns, Representative generalizations

- Let F and F' be two finite sets of function symbols. A signature substitution is an injective mapping $\eta : F \rightarrow F'$. It is extended to a function $\eta : Term(F,V) \rightarrow Term(F',V)$ by $\eta(f(t_1,\ldots,t_n)) = \eta(f)(\eta(t_1),\ldots,\eta(t_n))$ with $\eta(x) = x$ for all $x \in V$. Please note that in this case function symbols are purely syntactical elements. Even the arity of a symbol is not fixed but is defined only in the context of a concrete term.
- Let F be a finite set of function symbols, let $V = \{x_1, x_2, x_3, \ldots\}$ be a set of variable symbols and let $S = \{f_1, \ldots, f_k\}$ be another finite set of function symbols not necessarily

disjoint from F (Symbols from S are used as second order variables). The set $Term(F \cup S, V)$ is called the set of *term patterns* for $Term(F, V)$. A term $s \in Term(F \cup S, V)$ is called a *pattern* for a term $t \in Term(F, V)$ iff there exists a signature substitution $\eta : F \cup S \to F$ with $\eta(s) = t$.

- Assume $s, t \in Term(F \cup S, V)$. s is *more general* than t if s is a pattern for t, but not vice versa. s and t are*equivalent* if s is a pattern for t and t is a pattern for s.

- A term s is called *variable normalized* if the variables in s are picked from $(x_i)_{i \in \mathbf{N}}$ in the order of their appearance in the usual term notation.

- Let S and $F = F_f \uplus F_g$ be two sets of function symbols ($\uplus$ denotes the disjoint union, F_g are the function symbols to be generalized, while F_f are function symbols kept fixed). $t \in Term(F_f \cup S, V)$ is called a *most general pattern for $s \in Term(F, V)$ with respect to F_f and S*, if there exists a signature substitution $\eta : S \to F_g$ with $\eta(t) = s$ and there is no more general pattern with this property.

- Assume S and F as above. Let S be totally ordered according to some ordering $>_S$. A pattern t is called *operator normalized* if for all $f, g \in S$ $g >_S f$ implies that the first occurence of f precedes the first occurence of g.

- A pattern $t \in Term(F \cup S, V)$ is called *normalized*, if it is both variable normalized and operator normalized.

- Let again S and $F = F_f \uplus F_g$ be two sets of function symbols. We call the most general, normalized pattern for a term t with respect to F_f and S the *representative generalization of* t *with respect to F_f and S*. If $F_f = \{\}$ we just call it the *representative generalization* of t.

An important property of the representative generalization is its uniqueness.

Lemma 1 : Properties of representative generalizations

- For given sets F_f, S and a given $t \in Term(F, V)$ the representative generalization is unique. We write $\texttt{repgen}(t, F_f, S)$ to denote the representative generalization of t with respect to F_f and S.

- The representative generalization of a term can be computed effectively.

- The representative generalization is idempotent, that is:
  $\texttt{repgen}(\texttt{repgen}(t, F_f, S), F_f, S) = \texttt{repgen}(t, F_f, S)$.

We want to normalize not only terms, but also equations. To this end, we first define a total ordering on the term structure. For efficiency reasons this ordering makes use of a value already pre-calculated for all terms in DISCOUNT[1]: The weight of a term.

Definition 4 : Term weight
Let $t \in Term(F, V)$ be a term. The weight $W(t)$ is calculated as follows:

$$W(x) = 1 \text{ for all } x \in V \quad \text{and} \quad W(f(t_1, \ldots, t_n)) = 2 + \sum_{i=1}^{n} W(t_i)$$

[1] DISCOUNT uses this weight to cut down the search space for possible matches in simplification steps.

This weight is used in the definition of a total ordering on terms:

Definition 5 : The ordering $>_{pat}$
Let F a be a finite set of function symbols, let V be an enumerable set of variable symbols. Let $>_F$ and $>_V$ be total orderings on F and V, respectively. $>$ is the usual ordering on $\mathbf{N}$. Assume $s, t \in Term(F, V)$. $s >_{pat} t$ iff

1. $s, t \in V$ and $s >_V t$ or
2. 1. does not hold and $W(s) > W(t)$ or
3. Neither 1. nor 2. hold and $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$ and $f >_F g$ or
4. None of 1., 2. and 3. holds and $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_m)$ and $(s_1, \ldots, s_n) >^*_{pat} (t_1, \ldots, t_m)$

where $>^*_{pat}$ is the lexicographical extension to $>_{pat}$. The ordering is extended to equations as follows:

- Assume $s_1{=}t_1, s_2{=}t_2 \in (Term(F \cup S, V) \times Term(F \cup S, V))$. $s_1{=}t_1 >_{pat} s_2{=}t_2$ iff $(s_1, t_1) >^*_{pat} (s_2, t_2)$.

$>_{pat}$ can be computed very efficiently, as the pre-calculated weights usually lead to a very quick decision and a recursive descent can be terminated early. It is a total ordering on patterns.

Lemma 2 : $>_{pat}$ is total
Under the assumptions in definition 5, $>_{pat}$ is a total ordering on $Term(F \cup S, V)$.

This ordering can be extended to a quasi-ordering on the term structure:

Definition 6 : The quasi-orderings $\geq_{norm}$ and $\geq_{norm_0}$
Let $F \cup S$ be a finite set of function symbols with F_f[2] $\subseteq F$, let V be an enumerable set of variable symbols with orderings $>_{F \cup S}$ and $>_V$ as in definition 5. $>_{pat}$ is defined as above. Assume $s, t \in Term(F, V)$.

- $s \geq_{norm} t$ iff

  $\texttt{repgen}(s, F_f, S) >_{pat} \texttt{repgen}(t, F_f, S)$ or $\texttt{repgen}(s, F_f, S) \equiv \texttt{repgen}(t, F_f, S)$.

- $s \geq_{norm_0} t$ iff

  $\texttt{repgen}(s, \{\}, S) >_{pat} \texttt{repgen}(t, \{\}, S)$ or $\texttt{repgen}(s, \{\}, S) \equiv \texttt{repgen}(t, \{\}, S)$.

- Let $\geq_1$ be either $\geq_{norm}$ or $\geq_{norm_0}$. The strict part of $\geq_1$ is defined as usual: $s >_1 t$ iff $s \geq_1 t$ and $t \not\geq_1 s$. We write $s \approx_1 t$ iff $s \geq_1 t$ and $t \geq_1 s$.

- Let again $\geq_1$ be either $\geq_{norm}$ or $\geq_{norm_0}$. Assume $s_1{=}t_1, s_2{=}t_2 \in (Term(F \cup S, V) \times Term(F \cup S, V))$. $s_1{=}t_1 \geq_1 s_2{=}t_2$ iff $s_1 >_1 s_2$ or $s_1 \approx_1 s_2$ and $t_1 \geq_1 t_2$.

[2]As before F_f denotes the function ymbols to be kept fixed in a generalization process.

Note that these orderings require the assumptions made above. In particular, there exists a different ordering $\geq_{norm}$ for each F_f even if the other parameters are kept constant. For brevities sake we usually assume $\geq_{norm}$ to be implicitly defined with respect to the currently implied F_f.

The quasi-ordering $\geq_{norm}$ has two important properties.

Lemma 3 : Properties of $\geq_{norm}$
Assume F, F_f, V, S, s, t as in definition 6.

- $\geq_{norm}$ is total. For all $s, t \in Term(F, V)$ either $s \geq_{norm} t$ or $t >_{norm} s$.
- $s \approx_{norm} t$ implies that s and t have the same representative generalization with respect to F_f and S.

With these definitions and results we can lift the concept of representative generalizations to equations:

Definition 7 : Representative generalizations for equations
Let $s{=}t$ be an equation from $(Term(F,V) \times Term(F,V))$ and assume F, F_f, S and V as above. Let h be a new, binary function symbol not an element in any of these sets. $s'{=}t' \in (Term(F \cup S, V) \times Term(F \cup S, V))$ is a representative generalization for $s{=}t$ with respect to F_f, if

1. $s' \geq_{norm} t'$ and
2. $h(s', t')$ is a representative generalization for $h(s, t)$ with respect to $F_f \cup \{h\}$.

As above, we drop the clause *with respect to* F_f if $F_f = \{\}$.

Lemma 1 also holds for representative generalizations of equations. Note that $>_{norm}$ is also defined for equations. As this ordering reduces to $>_{pat}$ on representative generalizations, it is total on this subclass of terms.

Lemma 4 : $>_{norm}$ and representative generalizations
$>_{norm}$ is total on representative generalizations. Assume s and t are two representative generalizations with respect to some S and F_f, and $>_{norm}$ is defined for the same S and F_f. Then either $s >_{norm} t$ or $t >_{norm} s$ or $s \equiv t$.

With these definitions we can solve the problems pointed out at the beginning of the chapter.

5.3 Information To Be Stored

We want to store only some of the information contained in a complete PCL listing. As the knowledge is organized by (generalized) equations, there are two facts that will be recorded for each occurence of a term pair: The number of references made to this equation in the proof (a measure for the importance of the term pair), and the longest inference chain leading from this fact to the goal (an indication for the position of the fact in the proof process).

Definition 8 : The functions `references` and `goal_dist`
Remember that a PCL listing can be seen as representing a directed proof graph (compare section 2.4). Let *step* be the pcl-step at a node of this graph.

- `references`(*step*) is the number of outgoing links of this step in the pruned graph.
- `goal_dist`(*step*) is the maximum length (number of links traversed) of a path from *step* to the final step in the PCL listing.

As many proofs are collected in a case or in the global knowledge, a list of proofs and accompanying information is kept for each (generalized) equation. Thus, an entry for a single equation has the following structure: A generalized equation, followed by a colon and then a comma-separated list of occurence-descriptors. Each occurence-descriptor consists of the name of the example the equation occured in, followed by a tuple (`references`(*step*), `goal_dist`(*step*)) if *step* is the PCL step the equation occured in.

For the different domains we also want to store the fixed part of the signature – the function symbols appearing in the goal or in the specification, respectively.

Definition 9 : `sig` and `arity`

- Assume $T \subseteq Term(F,V)$. $\mathtt{sig}(T)$ is the set of all function symbols occuring in terms from T.
- $\mathtt{sig}(\{s_1{=}t_1, s_2{=}t_2, \ldots, s_n{=}t_n\}) = \mathtt{sig}(\{s_1, t_1, s_2, t_2, \ldots, s_n, t_n\}$.
- Assume again $T \subseteq Term(F,V)$. Further assume $f \in \mathtt{sig}(T)$. $\mathtt{arity}(f,T)$ yields the arity of f in the terms from T if f has a fixed arity. It produces an error notification if it is not used in terms from t or if it is used with varying arities.
- `arity` is extended to allow sets of equations as a second argument in the same way `sig` is.

5.4 Goal Domains

Domains (or cases) dealing with a common goal are closest to the data structures in classical EBG. Data about all proofs with the same representative generalization of the goal are collected into a single domain. The facts are generalized in a way that maintains their relationships to the goal. A proof is inserted into the goal-indexed part of the data base as follows:

- Assume $s{=}t$ is the goal of a proof and $E = \{s_1{=}t_1, s_2{=}t_2, \ldots, s_n{=}t_n\}$ is the set of all equations used to proof the goal. Assume further that $\mathtt{info}_i$ contains the list of occurence-descriptors associated with $s_i{=}t_i$.
- Further assume $S = \{f_1, f_2, \ldots, f_m\}$ and $V = \{x_1, x_2, \ldots, x_m\}$ with $f_i >_F f_j$ and $x_i >_V x_j$ if $i > j$ on $\mathbf{N}$.
- Calculate $s'{=}t' := \mathtt{repgen}(s{=}t, \{\}, S)$. Find the signature substitution η mapping $s{=}t$ onto $s'{=}t'$ and let $F_f = \mathtt{sig}(\{s'{=}t'\})$.

- Search the knowledge base for a goal domain with the goal $s'{=}t'$. If such a domain exists, let D be the existing knowledge tree from this domain, otherwise initialize a new domain and let D be an empty AVL tree. Calculate the (partial) signature for the new domain as the list containing $(f_i, \texttt{arity}(f_i, \{s', t'\}))$ for all $f_i \in \texttt{sig}(\{s', t'\})$.
- For each equation $s_i{=}t_i$ in E insert the tuple $(\texttt{repgen}(\eta(s_i{=}t_i), F_f, S\backslash F_f), \texttt{info}_i)$ into D, using the first part of the tuple as the key. If the key already exists in the tree, simply append $\texttt{info}_i$ to the second component of the entry.

After the application of this algorithm the AVL tree contains all relevant data about the equations used in the proof. In the same way a second tree is built, containing only the equations our lemma detection algorithms (compare section 2.4 and see [DS94a]) have selected as particularly important. As mentioned above, the AVL trees are stored one level after the other, to allow a reconstruction with the faster binary insert. Additional administrative data is stored with the domain. A goal domain's structure can be described by the following syntax:

```
Domain: <ident>
Type: Goal
Signature: <ident>:<num>[,<ident>:<num>]*
Examples: <ident>[,<ident>]*
Goals: <term> = <term>
Lemmas:
   [<term> = <term>:<ident>(<num>,<num>)[,<ident>(<num>,<num>)]*]*
Facts:
   [<term> = <term>:<ident>(<num>,<num>)[,<ident>(<num>,<num>)]*]*
```

<ident> is defined as usual, <num> is a nonempty sequence of digits. <term> describes a term with the usual syntax, briefly described in section 2.2.1. The signature describes only the function symbols used in the goal. These are used in a uniform way throughout the data. Accompanying each function symbol in the signature is its arity.

Of course the goal, the lemmas, and the other equations (or *facts*) are representative generalizations as mentioned above, using not arbitrary identifiers, but only identifiers from $\{\texttt{f1}, \texttt{f2}, \ldots, \texttt{fn}\}$[3] as function symbols and identifiers from $\{\texttt{x1}, \texttt{x2}, \ldots\}$ to represent variables. The lists stored with the lemmas and the facts contain the examples the equation was used in, each followed by the number of references and the goal distance.

The first half of the domain description, up to, but not including the keyword `Lemmas`, is called the *head* of the domain description. This part is stored in the `goaldoms`-file for the fast identification of a goal domain.

5.5 Specification Domains

Specification domains provide data for the formation of a strong general knowledge representation in a given equational theory. Data from all proofs in the same theory is collected in a

[3] Here we use the actual notation implemented by the programs, instead of the usually more convenient $\{f_1, f_2, \ldots, f_n\}$.

single structure. The implementation is very similar to the case of goal domains:

- Assume $\{u_1{=}v_1, u_2{=}v_2, \dots, u_m{=}v_m\}$ is the set of axioms used in a proof and $E = \{s_1{=}t_1, s_2{=}t_2, \dots, s_n{=}t_n\}$ is the set of all equations used to proof the goal. Assume again that $\mathtt{info}_i$ contains the list of occurence-descriptors associated with $s_i{=}t_i$.
- Assume $S = \{f_1, f_2, \dots, f_k\}$ and $V = \{x_1, x_2, \dots, x_l\}$ with $f_i >_F f_j$ and $x_i >_V x_j$ if $i > j$ on $\mathbf{N}$. Assume further that f is a new function symbol with arity $2m$ not used in the proof.
- Search the knowledge base for a similar specification domain, namely one with axioms that are, modulo function symbol renaming, identical to $\{u_1{=}v_1, u_2{=}v_2, \dots, u_m{=}v_m\}$[4]. The actual implementation speeds this search up by sorting the axioms using $>_{norm_0}$ and comparing normed specifications.
  - If such a domain exists, let D be the existing knowledge tree from this domain. Find the signature substitution η mapping the new specification onto the old one.
  - Otherwise, initialize a new domain and let D be an empty AVL tree. Let $t = f(u_1, v_1, u_2, v_2, \dots, u_m, v_m)$. Calculate $t' := \mathtt{repgen}(t, \{\}, S)$. Find the signature substitution η mapping t onto t'.
- Let $F_f = \mathtt{sig}(\eta(\{u_1{=}v_1, u_2{=}v_2, \dots, u_m{=}v_m\}))$.
- If the domain has been newly created, calculate the (partial) signature for the new domain as the list containing $(f_i, \mathtt{arity}(f_i, \eta(\{u_1{=}v_1, u_2{=}v_2, \dots, u_m{=}v_m\})))$ for all $f_i \in F_f$.
- For each equation $s_i{=}t_i$ in E insert the tuple $(\mathtt{repgen}(\eta(s_i{=}t_i), F_f, S \backslash F_f), \mathtt{info}_i)$ into D, using the first part of the tuple as the key. If the key already exists in the tree, simply append $\mathtt{info}_i$ to the second component of the entry.

As previously, the AVL tree built in this procedure contains all relevant data about the equations used in the proof. Again, two trees are built, one each for lemmas and arbitrary facts. The syntax of the a domain description looks like this:

```
Domain:  <ident>
Type:  Specification
Signature:  <ident>:<num>[,<ident>:<num>]*
Examples:  <ident>[,<ident>]*
Axioms:  [<term> = <term>]*
Lemmas:
   [<term> = <term>:<ident>(<num>,<num>)[,<ident>(<num>,<num>)]*]*
Facts:
   [<term> = <term>:<ident>(<num>,<num>)[,<ident>(<num>,<num>)]*]*
```

The remarks on the syntax in the previous section do of course apply here as well. In particular, the *head* up to, but not including the keyword `Lemmas` is duplicated separately in `specdoms`.

[4]Checking for semantic equivalence might seem an attractive idea. However, this problem is, of course, undecidable. Moreover, our experience suggests that the completion process can be heavily influenced by the exact syntax of the problem description.

5.6 Domain Independent Data

Data not bound by a goal or a specification can be dealt with even more simply and efficiently. As facts from all the proofs in the knowledge base are collected in a single AVL tree, this tree can grow to quite some size. We use two separate files to keep the overhead for the data access small. One file contains data from all the equations used in the proofs, the other one only data about the lemmas. Each of the two files simply contains the usual statistical data and the facts from the proofs. The insertion of a new proof proceeds as follows:

- Once more assume that $E = \{s_1{=}t_1, s_2{=}t_2, \ldots, s_m{=}t_m\}$ is the set of all equations (or all the lemmas) used to proof the goal and that $\texttt{info}_i$ contains the list of occurence-descriptors associated with $s_i{=}t_i$.
- Assume $S = \{f_1, f_2, \ldots, f_k\}$ and $V = \{x_1, x_2, \ldots, x_l\}$ with $f_i >_F f_j$ and $x_i >_V x_j$ if $i > j$ on $\mathbf{N}$. Let D be the AVL tree containing the previous knowledge.
- As before, for each $s_i{=}t_i$ in E insert $(\texttt{repgen}(s_i{=}t_i, \{\}, S), \texttt{info}_i)$ into D, using the first part as the key and appending the $\texttt{info}_i$ to a possibly existing entry.

This process is executed once for the complete data and once for the lemmas, yielding two files with the following syntax:

```
Brain:  <ident>
Type:  Complete|Pruned
Examples:  |<ident>[,<ident>]*
Facts:
   [<term> = <term>:<ident>(<num>,<num>)[,<ident>(<num>,<num>)]*]*
```

Chapter 6

Reusing Term Pair Structures

The first and obvious idea to use the stored data is to employ it in the straightforward evaluation of term pairs with a known structure. This is very similar to the learning process humans experience. A number of results in each domain are recognized as *of previous value*, and are prefered in dealing with related problems. This approach is likely to result in a very good initial knowledge representation and will facilitate the reuse of existing proofs. However, our results have shown that the extrapolative power of a learning prover using this approach is rather weak.

6.1 Knowledge Retrieval – The Experts

The experts resulting from the simple reuse approach mirror the structure of the knowledge base. Three different experts are suggested, each working with one of the three different parts of the knowledge base. The access to the stored data is strictly analogue to the insertion process described previously.

6.1.1 Using Goal Knowledge – `goal_bound_learn`

The `goal_bound_learn` evaluation strategy uses the knowledge collected in the goal domains. In an initialization phase the expert is searching the knowledge base for data pertaining to the goal at hand. As in the previous chapter, assume $S = \{f_1, f_2, \ldots, f_n\}$, and also assume $s{=}t$ to be the goal handed to the proof system. The algorithm used in the initialization phase of the expert can be described as follows:

- Calculate $s'{=}t' = \texttt{repgen}(s{=}t, \{\}, S)$. Find the signature substitution η mapping $s{=}t$ onto $s'{=}t'$. Set $F_f := \texttt{sig}(\{s'{=}t'\})$.
- Search the domain heads in the file `goaldoms` for a domain with the same goal. Due to the uniqueness of the representative generalization, a simple syntactical check is sufficient, making the search very efficient.
- If no suitable domain is found in the knowledge base, the expert will default to another strategy – either the *add* strategy or the evaluation using global knowledge only. If a matching domain is found, the relevant knowledge (either all equations or the lemmas,

depending on the users preference) is read from the appropriate file and the AVL tree is reconstructed. In this case η, F_f and the tree are stored for use in the retrieval of data for the evaluation of critical pairs.

If a critical pair $u{=}v$ is handed to the evaluation strategy, it accesses the knowledge in the AVL tree to retrieve the entry about equivalent pairs, following these instructions:

- Build $\mathtt{repgen}(\eta(u{=}v), F_f, S\backslash F_f)$ and use this as the key for a binary search in the AVL tree.
- If an appropriate entry is found, use the data stored with the pair for an evaluation. Otherwise, use a default strategy[1].

The exact interpretation of the retrieved knowledge is the same for all three knowledge access methods and is discussed in section 6.1.5.

6.1.2 The spec_bound_learn Strategy – Knowledge About Specifications

In a similar way we make use of the knowledge stored in the specification domains. However, the process is much more involved, as the essence of a specification is very difficult to capture in a unique representation. Moreover, simple syntactical identity is not very satisfactory for the specifications. Fortunately, one solution to these problems has already been implemented for DISCOUNT. Reusing the existing functionality of the planning component's domain detector enables the initialization phase to proceed in the following steps:

- For each specification domain in the knowledge base the set of defining equations (axioms) $\{u_1{=}v_1, u_2{=}v_2, \ldots, u_n{=}v_n\}$ is compiled into the appropriate form and handed to the existing domain recognition component (see [DK94] for more details).
- The algorithm of this component successively applies all possible signature substitutions between the domain's signature and the current problem's signature to the defining equations of the domain. For each substitution, it tries to prove the equations in DISCOUNT's current system, using only simplification and subsumption tests[2].
- If all the defining equations for a domain have been proven valid, the signature substitution η mapping the domain's specification onto the current problem is reconstructed from the internal data structures of the recognition component and F_f is set to $\mathtt{sig}(\eta(\{u_1{=}v_1, u_2{=}v_2, \ldots, u_m{=}v_m\}))$. The relevant domain knowledge is then read from the appropriate file and, as above, used to build the AVL tree.
- Depending on the users preference, further domains can be checked. If another domain can be proved valid with the same signature substitution, its knowledge is added to the tree. Different signature substitutions are not allowed, as they would endanger the integrity of the knowledge representation.

[1]As known pairs are to be prefered over unknown pairs, a hefty penalty is added to the evaluation by either *add* or the global knowledge strategy. The penalty is sufficient to ensure that known pairs are always prefered to unknown ones.

[2]This test is, of course, not complete, but it ensures termination and reasonable efficiency.

- Finally, if no matching domain was found, a default strategy is used as above.

As above, η, F_f and the AVL tree are used in the access of the stored knowledge. A critical pair to be evaluated is treated exactly as in the previous case.

6.1.3 Using Global Knowledge – `global_learn`

The retrieval of data about pairs from the global knowledge store is the most simple of the retrieval operations introduced in this chapter. In the initialization phase the knowledge is read from the relevant file in the knowledge base – either `cdata` for the complete information or `pdata` if only the lemmas are desired. As before, the facts are organized into an AVL tree. To retrieve possible data about an equation $s{=}t$, the representative generalization `repgen`$(s{=}t, \{\}, S)$ is used as the key in a binary search for the tree. The evaluation of the equation then proceeds as in the other two cases, with the exception that, of course, only *add* is available as a backup strategy in case the pair was not found.

6.1.4 Fairness

Even without detailed information about the use of the information retrieved from the knowledge base the fairness of the learning experts can be shown. The following theorem holds:

Theorem 5 : Fairness of the strategies
The strategies `goal_bound_learn`, `spec_bound_learn` and `global_learn` are fair if the backup strategies used to evaluate pairs not found in the knowledge base conform to the conditions required in theorem 4.

Proof: According to theorem 4 we only need to show that the evaluation functions yield integer values greater than or equal to a base value and that only a finite number of pairs is mapped onto the same integer value.

- The functions yield integers by definition.
- The terms occuring in the DISCOUNT system are build from a finite set of symbols. As the representative generalization of an equation has the same number of symbols as the equation, only a finite number of equations share the same representative generalization. As the AVL trees are finite, only a finite number of equations are evaluated using data from the knowledge base. Any finite set has a minimum. Call this minimum a and call the minimum weight assigned by the backup strategy b. Then the evaluation function yields only values greater than or equal to $min\{a, b\}$.
- As only a finite number of equations is evaluated using the tree, only finitely many equations are mapped onto the same value by the AVL tree. The same holds for the backup strategy and thus for the evaluation function as a whole.

6.1.5 Evaluating Previous Experience

If the key generated from the critical pair has been found in the tree, the associated data items can be retrieved for computing the final evaluation.

- The number of proofs in which equivalent facts were used can be inferred from the size of the occurence descriptor list.
- The maximum distance between the first occurence of an equation and the final proof step is recorded for each proof. For the evaluation this information is averaged over all the proofs.
- The total number of references to the equation is stored for each of the previous proofs. This information is summed up to yield an overall total.
- Finally, the critical pair to be evaluated is still available.

The total number of proofs a fact was useful in as well as the total number of references to the fact can be used as straightforward indicators for its usefulness. The terms in the equation can be evaluated by a standard strategy and thus contribute to the overall verdict. However, the influence of the distance to the final proof step is less obvious. We have considered four different (and mutually exclusive) strategies to use this information in a judgment for the quality of an equation, each of which can be rationalized.

- First, an *aggressive* strategy might try to go directly to the goal and prefer equations with a low distance to the final step.
- In order to keep the completion as long as possible in an area that is covered by the available knowledge it might be useful to prefer facts that have a maximal distance to the goal. This *conservative* approach aims at slowly building a strong knowledge representation.
- A more active approach is the *opportunistic* strategy. It tries to mimic the original proof as close as possible by prefering pairs that have a distance close to the one that has been evaluated previously. Thus, some equations might fall behind and the prover follows only one thread towards the goal.
- The opportunistic strategy is vulnerable to random critical pairs that can throw it of its track. A more stable behaviour can be achieved if an *adaptive* strategy allows the last critical pairs to only slowly shift the focus of the completion.

Each of these approaches seems viable. We have implemented all four strategies. Results from the next sections will show which of the arguments is valid.

The single influence factors are combined in a weighted sum to give the final verdict. The weighting factors are provided by the user. Now assume $s{=}t$ to be the critical pair to be evaluated, let $total$ be the sum of all references in this pair, let $proofs$ be the number of proofs the fact contributed to, let $average = \lfloor total/proofs \rfloor$ be the average number of references per proof and let $goal\_dist$ be the average distance of the equation to the final proof step. Then the value of the pair is computed as follows (remember that a low value designates a desirable pair):

$$value := (W(s) + W(t)) + (-const * (goaldist\_influence * goal\_w$$
$$+ total * total\_w + proofs * proofs\_w + average * average\_w))$$

In this formula $goal\_w$, $total\_w$, $proofs\_w$ and $average\_w$ are (user-defined) constants from $\mathbf{N}$ determined at the start of the proof process. $const$ is a fixed value set to 20 in the implementation, thus ensuring that the influence of the learned knowledge usually prevails. $goaldist\_influence$ represents the evaluation of the average distance to the final proof step. It is calculated as follows:

- For the aggressive strategy, $goaldist\_influence$ is simply set to $-goal\_dist$, thus prefering pairs with a small distance.
- For the conservative strategy, $goaldist\_influence$ equals $goal\_dist$.
- For the opportunistic strategy we need a global state that is maintained beyond the scope of one call to the evaluation function. This $goal\_state$ is initialized to the greatest distance of any known pair. In the evaluation the value of $goaldist\_influence$ is set to $-|goal\_state - goal\_dist|$ and the value of $goal\_state$ is updated to $goal\_dist$.
- Finally, for the adaptive strategy $goaldist\_influence$ is set to $-|goal\_state - goal\_dist|$ and $goal\_state$ is updated to $\lfloor (3 * goal\_state + goal\_dist)/4 \rfloor$, thus changing it more slowly as for the opportunistic strategy[3].

Now let us take one more look at the formula for the value of a pair found in the knowledge base. The first element of the sum is equivalent to the weight assigned by the pure *add* strategy. It was included because preliminary tests showed that the learning component maps most equations onto only a small number of values and thus lacks the ability for a finer distinction. The second term of the sum represents the former experience about the pair. As we want this to be the more important influence, it is weighted with a reasonably large value (20 in our case). Finding good values for the different weights in the formula will be part of the test phase.

6.2 Results for the Sequential Case

While our ultimate goal is to integrate the learning experts into a distributed TEAMWORK team, the analysis of their performance is much easier for the sequential case. The influence of a single strategy can be studied isolated and without interference from other team members. In addition, more practical considerations also suggest an evaluation of the learning strategies in the sequential system. The sequential prover does not rely on critical timing for cooperation and its behaviour is therefore not influenced by the load imbalances which are not avoidable in a multi-user, multi-tasking environment. Results are more reliable and easier to reproduce. Moreover, a sequential test run ties up less system resources, allowing multiple test runs at the same time.

6.2.1 Testing Methods

We will nearly exclusively use a constant set of examples from the domain of lattice ordered groups for the evaluation of the learning experts. This domain offers a large number of

[3] The function for the updating is, of course, an instance of the more general formula $\lfloor (const * goal\_state + goal\_dist)/(const + 1) \rfloor$. However, the parameter space for the evaluation strategy is already enormous, so we did refrain from adding yet another parameter.

examples of varying difficulty. Appendix A describes it in some detail, including data about the performance of the conventional experts in the domain.

We will use three different modes for the test runs. In all modes DISCOUNT tries to prove all of the 86 selected examples. The difference is the composition of the knowledge base. *Explorative* or *autonomous* mode aims at simulating a stand-alone learning system without any influence by the user. In a completely closed-loop approach, the examples are handed to the prover in a fixed order. If a proof is found, exactly this proof is handed back to the learning component, otherwise the knowledge base is unchanged.

The second, *persistent* mode probably is the most realistic simulation of a proof system actually used by humans in a normal environment. In this case a proof is added to the knowledge base regardless of whether the proof was found by the learning strategy or by any other means. Note that there are still some examples for which *no* proof is available at all. In these cases the knowledge base is obviously not changed.

Finally, in *reproductive* mode the prover starts with a knowledge base stocked with all the knowledge available about the domain. The knowledge base is constructed at the beginning of the test and does not change during it. This simulates an environment where the same problems reappear multiple times after it has been solved once.

In each of the three modes we give the examples to the prover in the fixed order they appear in appendix A. The prover was allowed 1000 seconds of cpu time to solve each problem. If no proof was found in the time, the system was terminated and the proof run was counted as a failure. For each series of tests with a single expert and a fixed set of parameters we give the number of proofs found. We also give the accumulated run time over all examples, counting failures with 1000 seconds. This allows for a comparison between experts that proved the same number of examples and gives a qualitative impression for the performance of the expert on all examples. Times recorded are cpu time. They correspond, within only fractions of a second, to real time on an otherwise unused machine not forced into swapping.

The most difficult of the three situations is, of course, explorative mode. We will therefore use this mode for more detailed analysis and the tuning of the parameters. The other two modes will only be tested with a constant set of parameters to determine the performance of the prover under more favorable conditions.

Moreover, we will concentrate on the `global_learn` strategy. Test results will demonstrate that the strategies using only restricted sets of the data determined by the goal can further improve the behaviour in some cases.

6.2.2 Influence of the Goal Distance Evaluation

Table 6.1 shows the accumulated run times (in seconds) and the number of proved examples for the four different strategies to evaluate the distance to the final proof step. The `global_learn` expert was used in explorative mode, with the complete global knowledge available at each point. All parameters except for *goal_w* were set to zero. The same examples were proved in all 4 cases. We also included the time for the successful examples only, as the 28000 seconds corresponding to the 28 failures tend to confuse the small trend recognizable.

Obviously, the choice of the evaluation strategy for the goal distance of a pair is, at best, a

Strategy	**aggressive**	**opportunistic**	**adaptive**	**conservative**
Successes	58	58	58	58
Accumulated Time	28460.79	28442.53	28436.08	28436.45
Succ. time	460.79	442.53	436.08	436.45

Table 6.1: Results for different goal distance evaluation strategies

minor influence for the performance of the expert[4].

6.2.3 Other parameters

Table 6.2 shows the effect of the other parameters. As above we give both the total time for all examples and the time only for the successful proofs. Generally, only rather large values change the behaviour of the prover in any significant way – and the parameter *average_w* is without any measurable influence for the values we tested.

avarage_w	*proofs_w*	*total_w*	**Successes**	**Acc. Time**	**Succ. Time**
1	0	0	58	28449.98	449.98
2	0	0	58	28451.43	451.43
3	0	0	58	28450.22	450.22
10	0	0	58	28451.45	451.45
20	0	0	58	28451.21	451.21
0	1	0	58	28461.46	461.46
0	2	0	58	28462.69	462.69
0	3	0	58	28466.29	466.29
0	10	0	58	28601.31	601.31
0	20	0	58	28462.05	462.05
0	0	1	58	28454.21	454.21
0	0	2	58	28516.07	516.07
0	0	3	58	28429.63	429.63
0	0	10	58	28582.40	582.40
0	0	20	58	28445.43	445.43

Note: The parameter *goal_w* was set to 3 in this test, and the strategy was set to *aggressive*.

Table 6.2: `global_learn` for different parameters

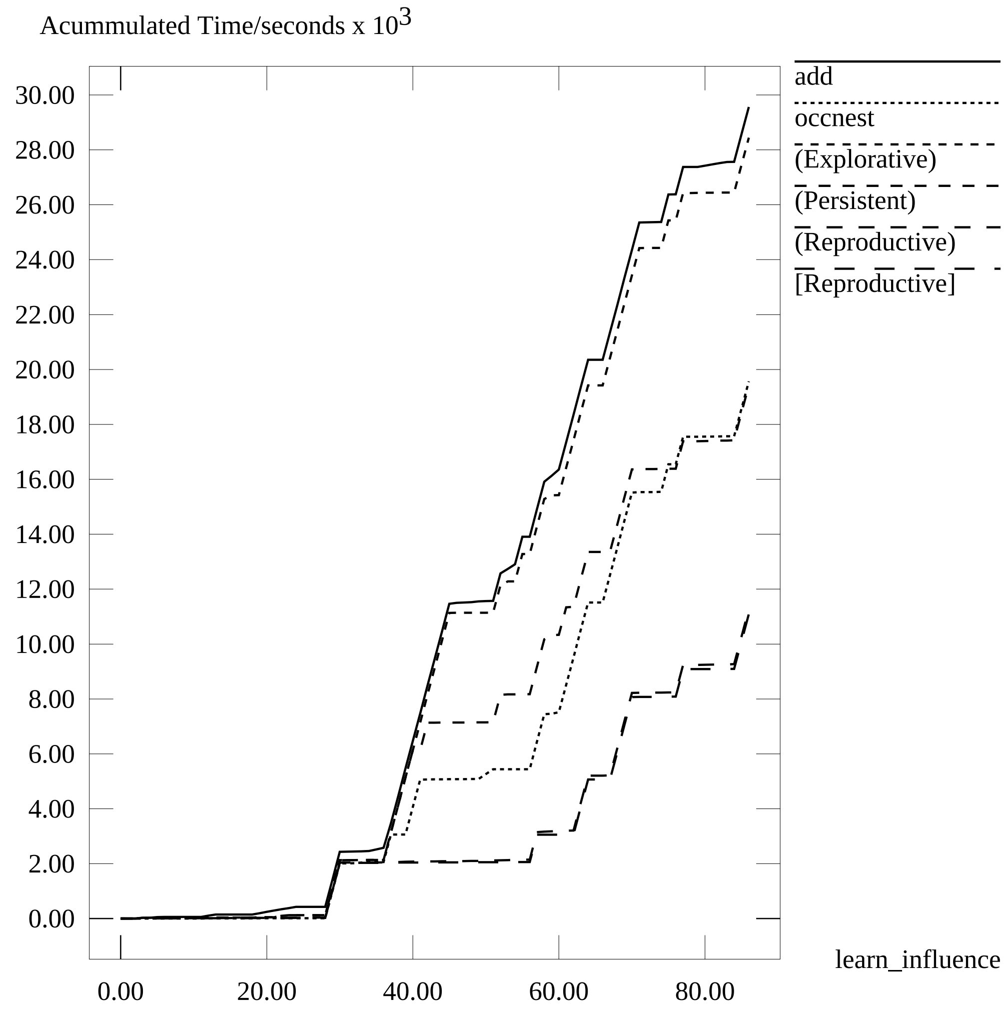
6.2.4 Detailed results

We will now take a closer look at the behaviour of the strategy in the different test modes. Tables 6.3 to 6.5 contains the run times for single examples for the `global_learn` strategy

[4]However, preliminary tests with an intermediate version showed that in a evaluation strategy ignoring the actual terms and relying *only* on the goal distance, significant changes can be observed.

in all three test modes. Also included is a reproduction run by the goal bound expert, and, for easier reference, the conventional strategies *add* and *occnest*.

Fig 6.1 visualizes the data from the tables. It shows accumulated runtimes versus number of processed examples (examples were given to the prover in the order they appear in the table). For failures (examples no proof was found for), 1000 seconds were assumed for the accumulated time.



Note: The entries in round braces belong to the `global_learn` strategy, while the entry in square braces belongs to `goal_bound`.

Figure 6.1: Comparative performance of `global_learn` and `goal_bound`

Examples	add	occnest	global_learn			goal_bound
			Explor.	**Persist.**	**Repro.**	**Repro.**
ax_antisyma	0.230	0.190	0.590	0.680	0.860	0.750
ax_antisymb	0.210	0.230	0.260	0.240	0.710	0.760
ax_glb1a	27.790	0.820	31.350	30.680	2.160	1.180
ax_glb1b	0.730	0.290	0.370	0.400	1.830	0.990
ax_glb1c	28.110	0.820	0.420	0.390	2.260	1.140
ax_glb1d	0.800	1.020	0.340	0.440	1.910	1.020
ax_glb2a	0.220	0.200	0.290	0.340	0.750	0.940
ax_glb2b	0.250	0.210	0.530	0.620	0.900	0.920
ax_glb3a	0.220	0.210	0.220	0.260	0.650	0.860
ax_glb3b	0.230	0.210	0.430	0.520	0.900	0.690
ax_lub1a	2.490	0.270	0.360	0.420	1.790	1.000
ax_lub1b	42.900	2.160	0.440	0.450	2.190	1.120
ax_lub1c	42.750	2.200	0.430	0.430	2.190	1.160
ax_lub1d	2.470	1.050	0.410	0.420	1.840	1.030
ax_lub2a	0.290	0.220	0.490	0.460	1.260	0.750
ax_lub2b	0.200	0.220	0.290	0.350	0.720	0.870
ax_lub3a	0.260	0.230	0.370	0.380	0.870	0.800
ax_lub3b	0.210	0.210	0.310	0.300	0.670	0.930
ax_mono1a	46.670	0.200	8.650	8.400	1.090	1.070
ax_mono1b	47.920	0.220	0.570	0.590	1.680	0.840
ax_mono1c	47.560	0.300	0.500	0.550	1.740	0.980
ax_mono2a	44.860	0.220	56.430	55.110	1.080	0.860
ax_mono2b	46.380	0.220	20.540	20.480	1.680	1.030
ax_mono2c	45.120	0.240	0.730	0.780	1.040	0.960
ax_refla	0.210	0.190	0.560	0.540	0.850	0.960
ax_reflb	0.160	0.170	0.490	0.580	0.900	0.970
ax_transa	0.460	1.450	0.450	0.400	1.900	1.180
ax_transb	0.430	1.450	0.450	0.430	1.880	1.160
distrnu	–	–	–	–	–	–
distrun	–	–	–	–	–	–
lat1a	6.550	0.270	1.470	1.510	1.050	1.050
lat1b	6.570	0.210	0.880	0.940	1.160	0.840
lat2a	9.310	24.160	2.910	3.140	1.050	1.060
lat2b	9.160	21.46	1.310	1.550	1.240	1.100
lat3a	56.670	0.250	1.260	0.710	1.710	0.960
lat3b	56.960	0.260	1.060	1.380	2.180	0.890
lat4	893.50	–	–	–	6.970	5.480

Table 6.3: Performance of global_learn and goal_bound in different test modes

Examples	add	occnest	global_learn			goal_bound
			Explor.	**Persist.**	**Repro.**	**Repro.**
p01a	–	0.480	–	–	8.490	1.090
p01b	–	0.460	–	–	8.610	1.080
p02a	–	–	–	–	5.720	1.220
p02b	–	–	–	0.830	2.350	1.140
p03a	–	4.400	–	–	3.040	1.210
p03b	–	2.690	–	2.490	2.940	1.200
p03c	–	2.760	–	1.770	2.920	1.240
p03d	–	4.370	–	1.290	3.020	1.240
p04a	34.220	2.000	0.920	0.960	2.000	1.180
p04b	9.700	1.930	2.110	1.610	2.310	1.110
p04c	9.790	1.900	1.180	1.580	2.290	1.130
p04d	33.780	2.020	0.560	1.090	2.000	1.190
p05a	7.930	177.35	0.980	2.850	7.860	0.960
p05b	7.960	177.92	0.950	1.370	7.910	1.030
p06a	–	0.630	–	–	7.900	0.940
p06b	165.810	0.320	136.20	17.260	7.810	0.870
p06c	167.190	0.310	1.070	1.460	7.830	0.860
p06d	–	0.570	–	1.500	7.840	1.060
p07	5.100	0.190	0.460	0.550	0.810	0.850
p08a	–	–	–	–	–	–
p08b	–	–	–	–	19.61	02.910
p09a	219.480	20.470	139.36	153.290	13.60	01.560
p09b	224.350	52.330	3.060	13.270	13.62	01.600
p10	–	–	–	–	4.870	3.770
p11	–	–	–	17.210	8.100	2.950
p12	–	–	–	–	–	–
p12x	–	–	–	–	–	–
p17a	0.280	0.200	0.650	1.240	1.250	0.910
p17b	0.250	0.210	0.350	0.670	0.740	0.940
p18	–	–	–	7.230	8.140	2.800
p19	–	–	–	–	–	–
p20	–	–	–	–	–	–
p20x	–	–	–	–	–	–
p21	–	19.520	–	8.120	8.470	3.480
p21x	7.890	0.300	1.420	1.550	1.730	1.250

Table 6.4: Performance of global_learn and goal_bound in different test modes (continued)

Examples	**add**	**occnest**	`global_learn`			`goal_bound`
			Explor.	**Persist.**	**Repro.**	**Repro.**
`p22a`	5.100	0.970	0.900	1.240	1.370	1.010
`p22b`	5.040	11.070	1.120	0.960	1.090	1.290
`p23`	–	–	–	7.910	8.200	8.060
`p23x`	4.770	0.260	1.040	1.390	1.150	0.960
`p33`	–	–	–	–	–	–
`p38a`	0.250	0.190	0.840	0.830	0.880	0.880
`p38b`	0.230	0.180	0.360	0.600	0.690	0.840
`p39a`	45.210	5.460	9.890	6.740	2.070	0.880
`p39b`	45.400	4.070	1.700	7.910	7.450	1.040
`p39c`	45.340	4.040	1.390	6.920	7.280	0.970
`p39d`	45.250	5.460	1.300	3.630	2.080	1.080
`p40a`	5.240	2.030	1.460	10.050	6.460	0.930
`p8_9a`	–	–	–	–	–	–
`p8_9b`	–	–	–	–	–	–
Successes	59	67	58	67	75	75
Failures	27	19	19	19	11	11
Accumulated Time	29564.4	19569.1	28445.43	19422	11266.2	11099.01

Note: The parameters used were identical in all cases: $goal\_w = 3$, an aggressive strategy for the evaluation of the distance to the goal, $total\_w = 20$, $average\_w = 0$ and $proofs\_w = 0$. The strategy **`goal_bound`** used **`global_learn`** as a backup for unknown pairs. All times are in seconds. A dash indicates that no proof was found within the 1000 seconds time limit.

Table 6.5: Performance of `global_learn` and `goal_bound` in different test modes (concluded)

6.2.5 Evaluation

The reuse strategies are not very powerful in explorative mode. However, they excel if they have access to experiences made by other experts. The result of `global_learn` in persistent test mode rivals *occnest*. For the reproduction of training proofs the experts are excellent. In a realistic environment, this strategies are a valuable addition to DISCOUNT's arsenal even for sequential proofs.

Chapter 7

Extrapolation by Abstraction – TETs and TATs

The results from the previous chapter show that the simple reuse of recognized term pair structures is very well suited to the task of proof reproduction. However, it also showed a serious lack in the extrapolative power of the experts. The learning component is of little help as soon as the known territory is left. This chapter will introduce a knowledge representation that allows for much more abstraction than the representative generalizations – the *term evaluation tree* or, shorter, *TET*. All the relevant information from the knowledge base is compiled into a single TET and is then used to evaluate the terms of the critical pairs that are to be judged.

7.1 Term Evaluation Trees

Contrary to the AVL trees used in the knowledge base and by the experts detailed in the last chapter, TETs do not contain term pairs or even terms. Instead, all the terms (with some accompanying information) are compiled into a single tree structure. In their current incarnation, TETs abstract from all properties of a term except for its structure and the arities of the involved function symbols.

A TET represents a number of possible alternatives for the structure of terms. Each of these alternatives starts with a function symbol with a different arity and is represented by a *term arity tree* or TAT. A TAT represents all the alternatives that start with a function with a fixed number of arguments. Each of these arguments can again be one of several alternatives and is represented by a TET. More exactly, TETs and TATs are defined by mutual recursion:

Definition 10 : TETs and TATs
Let $INFO$ be an arbitrary knowledge representation language[1].

- A term evaluation tree is a (possibly empty) unordered list of term arity trees.
- A term arity tree is a tuple $(arity, count, info, (tet_1, \dots, tet_{arity}))$, where the tet_i are non-empty term evaluation trees, $arity$ and $count$ are elements from $\mathbf{N}$ and $info \in$

[1]In our implementation we use $INFO = \mathbf{N}$ and store only a single integer value at each node.

$INFO$ contains some arbitrary information about the TAT. We call $info$ an *annotation* to the tree node, $arity$ gives the arity of the TAT.

- TET is the set of all term evaluation trees.

It is easy to see that the recursion is only well-founded if eventually all branches end in leave nodes that are TATs of arity 0. Only such finite TETs are of interest to us, as only finite TETs are needed to represent finite sets of terms. Terms are inserted into a TET by a recursive function.

Definition 11 : Inserting terms into TETs
Let $T = (tat_1, \ldots, tat_m)$ be a term evaluation tree, let $t \in Term(F)$ be a ground term with n principal subterms, $t \equiv f(t_1, \ldots, t_n)$. Assume $info \in INFO$ and let $M : INFO \times INFO \rightarrow INFO$ be an associative and Abelian function merging two elements from $INFO$. We call a tuple $(t, info)$ an annotated term and $info$ the annotation. $\circ$ represents the list constructor and $()$ the empty list. The function `tet_insert` $: TET \times (Term(F) \times INFO) \rightarrow TET$ is defined as follows:

- If none of the tat_i has arity n, $\texttt{tet\_insert}(T, (t, info)) :=$
  $(n, 1, info, (\texttt{tet\_insert}((), (t_1, info)), \ldots, \texttt{tet\_insert}((), (t_n, info)))) \circ T$

- Otherwise, there exists a tat_i of the form $(n, count, info', (tet_1, \ldots, tet_n))$. In this case $\texttt{tet\_insert}(T, (t, info)) := (n, count + 1, M(info, info'),$
$$(\texttt{tet\_insert}(tet_1, (t_1, info)), \ldots, \texttt{tet\_insert}(tet_n, (t_n, info))))$$
$$\circ(tat_1, \ldots, tat_{i-1}, tat_{i+1}, \ldots, tat_m).$$

`tet_insert` is extended to a function $TET \times (Term(F, V) \times INFO) \rightarrow TET$ by treating variables as function symbols of arity 0.

The function M merging two elements from $INFO$ is required to be both associative and Abelian. As a result, the TET emerging from a series of insertions does not depend on the particular order of the terms.

Lemma 5 : `tet_insert` commutes
Assume $T \in TET$, $t_1, t_2 \in Term(F, V)$ and $info_1, info_2 \in INFO$. Then the following holds.

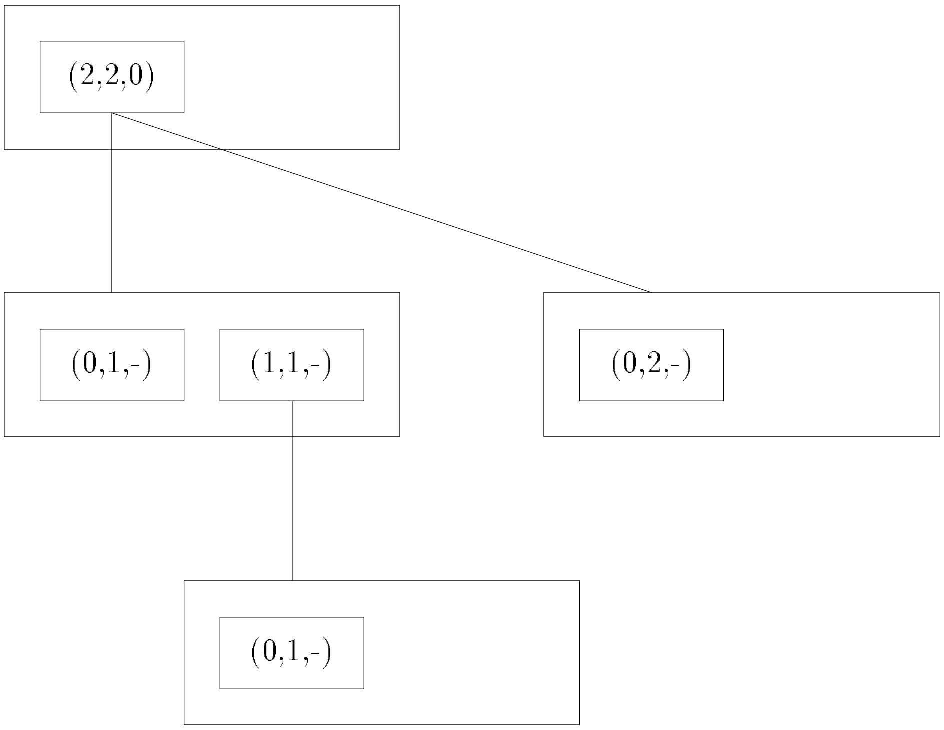
$$\texttt{tet\_insert}(tet\_insert(T, (t_1, info_1)), (t_2, info_2)) =$$
$$\texttt{tet\_insert}(tet\_insert(T, (t_2, info_2)), (t_1, info_1))$$

Due to this property of TETs it makes sense to speak of *the* TET representing a number of terms (with accompanying information).

Definition 12 : Representative TETs

- Let $D = \{(t_1, info_1), \ldots, (t_n, info_n)\} \subset Term(F, V) \times INFO$ be a set of terms with additional data. The *representative term evaluation tree* for D is the TET emerging if the tuples $(t_i, info_i)$ are successively inserted into an initially empty tree. We write $\texttt{rep\_TET}(D)$ to denote the representative TET for D.

- Let $D = \{(s_1{=}t_1, info_1), \ldots (s_n{=}t_1, info_n)\}$ be a set of equations with additional data. Then
  $\texttt{rep\_TET}(D) := \texttt{rep\_TET}(\{(s_1, info_1), (t_1, info_1), \ldots, (s_n, info_n), (t_n, info_n)\})$.



Note: The large rectangles represent TET nodes, while the smaller rectangles correspond to TAT nodes.

Figure 7.1: A small example TET

According to the previous lemma, the representative TET depends only on the actual data, not on the order of insertion. While the recursive definition of TETs is rather simple, the actual structure can become quite complex. Consider the simple example with $D = \{(f(x,y),-),(f(g(x),y),-)\}$ and a trivial function M for the merging of annotations defined by $M(-,-) = -$. Then

$$\texttt{rep\_TET}(D) = ((2,2,-,(((0,2,-,())),((0,1,-,()),(1,1,-,((0,1,-,()))))))))$$

Fig. 7.1 shows the structure of this TET. We now want to use TETs for the evaluation of terms. This is done by mapping the nodes of the term onto the corresponding notes in the TET and collecting the information stored in these nodes.

Definition 13 : Term evaluation using TETs
Now assume $INFO = \mathbf{N}$ and $M : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ given by $M(info_1, info_2) = info_1 + info_2$. Further assume $T = (tat_1, \ldots, tat_m) \in TET$ and $t = f(t_1, \ldots t_n) \in Term(F,V)$ with the top symbol F of arity n[2]. Let $E : (F \cup V) \times \mathbf{N} \times \mathbf{N} \times \mathbf{Z} \to \mathbf{Z}$ be a function for the evaluation of a single node of the term. Then the three functions $\texttt{eval}_\texttt{W} : \texttt{Term(F,V)} \times \texttt{TET} \times \texttt{Z} \to \mathbf{Z}$, $\texttt{eval}_\texttt{C} : \texttt{Term(F,V)} \times \texttt{TET} \to \mathbf{Z}$ and $\texttt{eval}_\texttt{I} : \texttt{Term(F,V)} \times \texttt{TET} \to \mathbf{Z}$ are defined as follows.

[2]Keep in mind that we treat variables as function symbols with arity 0.

- If there is no tat_i with arity n in T, then
  - $\texttt{eval}_{\texttt{W}}(t,T,k) = unk\_penalty * W(t) * k$
  - $\texttt{eval}_{\texttt{C}}(t,T) = 0$
  - $\texttt{eval}_{\texttt{I}}(t,T) = 0$
- Otherwise there is a $tat_i = (n, count, info, (tet_1, \ldots, tet_n))$ in the TET. Then
  - $\texttt{eval}_{\texttt{W}}(t,T,k) = E(f, count, info, k) + \sum_{i=1}^{n} \texttt{eval}_{\texttt{W}}(t_i, tet_i)$
  - $\texttt{eval}_{\texttt{C}}(t,T) = count + \sum_{i=1}^{n} \texttt{eval}_{\texttt{C}}(t_i, tet_i)$
  - $\texttt{eval}_{\texttt{I}}(t,T) = info + \sum_{i=1}^{n} \texttt{eval}_{\texttt{I}}(t_i, tet_i)$

The constant *unk_penalty* represents a penalty multiplied with the weight of subterms not represented in the TET at all.

7.2 The Experts `1tet_learn` and `2tet_learn`

With the above definitions we can now define the two TET-based experts developed for DISCOUNT. They differ only in the evaluation function $\texttt{eval}_{\texttt{W}}$ and in the final use of the single term evaluations in the evaluation of a critical pair. Both strategies share the following initialization routine:

- Let $D = \{s_1{=}t_1, \ldots, s_n{=}t_n\}$ be the set of all equations stored in the global part of the knowledge base (represented by the file `cdata`).
- Let $proofs_i$ be the number of proofs the equation $s_i{=}t_i$ was used in.
- Calculate to the representative TET for the knowledge read from the base. More exactly, $T := rep\_TET(\{(s_1{=}t_1, proofs_1), \ldots, (s_n{=}t_n, proofs_n)\})$. $TET\_size := 2{*}|D|$ is the number of terms inserted into the TET.

T and TET_size are used in the evaluation of the critical pairs. Two different approaches are used. The first strategy allows single term nodes[3] to contribute to the overall judgment in an arbitrary way, only limited by the knowledge found in the TET. A single term node may contribute any weight, even a negative one, to the total weight for the critical pair[4].

[3] A *term node* is a node in the tree representation of a term. It carries a function symbol and has the arguments as its descendants.

[4] Again, keep in mind that a low result value means that the corresponding critical pair will be prefered.

Definition 14 : The evaluation strategy `1tet_learn`
Let T be a term evaluation tree and let $s{=}t$ be a critical pair to be evaluated. Assume $E : (F \cup V) \times \mathbf{N} \times \mathbf{N} \times \mathbf{Z} \to \mathbf{Z}$ defined by $E(f, count, info, k) = 0$ for the definition of $\mathtt{eval_W}$ (compare definition 13). Then the value of the critical pair is given by

$$value := W(s) + W(t) + \mathtt{eval_W}(s, T, 1) + \mathtt{eval_W}(t, T, 1) - learn\_influence$$
$$*\left\lfloor \frac{\sqrt{proofs\_w * (\mathtt{eval_I}(s, T) + \mathtt{eval_I}(t, T))}}{(TET\_size * \mathtt{eval_C}(s, T) + 1 + \mathtt{eval_C}(t, T) + 1)} \right\rfloor$$

learn_influence and *proofs_w* are user-defined constants, as is *unk_penalty*, which is used indirectly via the function $\mathtt{eval_W}$.

The above formula certainly needs some explanations. As with the experts from the previous chapter, the normal weights of the terms are included for fine-tuning. The next two parts of the sum add a large penalty for parts of the terms that do not have corresponding nodes in the TET. The final part contains the condensed knowledge from the TET. *learn_influence* determines the importance of this for the total weight. The fraction multiplied with *learn_influence* relates the number of terms inserted, the number of terms in successful pairs that share structures with the terms to be evaluated, and the number of proofs such terms were successful in. The square root was added as the result of first experiments that showed that the straight product grew to fast with the number of terms inserted, making the prover extremely conservative and limited to known areas. This is still a problem for large amounts of data. However, as the next sections will show, this function results in a very strong expert. It also is a fair evaluation strategy:

Theorem 6 : Fairness of `1tet_learn`
The strategy `1tet_learn` is fair.

Proof: As before, we only need to show that the evaluation function yields integer values greater than or equal to a base value and that only a finite number of pairs is mapped onto the same integer value.

- The functions yields an integer value by definition.
- The first four terms in the sum for *value* are positive by definition. The argument of the square root in the fraction is the product of a finite sum and a constant value. Let *max* be the maximum of all the annotations in the TET. Then the two sums are smaller than or equal to $TET\_size * max$. Consequently, the numerator also has an upper bound. The divisor is guaranteed to be greater than or equal to 2. Thus, the total fraction has an upper bound and *value* has a lower bound.
- The fraction has not only an upper bound but is also guaranteed to be no smaller than 0. Thus, it can only have a finite number of results. Both W and $\mathtt{eval_W}$ map only a finite number of terms onto the same value as they both grow with the number of symbols in a term (for terms above a certain size for $\mathtt{eval_W}$) and, for a given finite signature, only a finite number of terms exist at a given size. Thus the sum also maps only a finite number of pairs onto the same value.

The second strategy results from a slightly different view on the problem. The impact a single term node can have is limited – no term node may actually improve the overall weight. However, a term node belonging to a term structure often found in successful critical pairs is much less of a liability to the pair than a node not belonging to a successful structure.

Definition 15 : The evaluation strategy `2tet_learn`
Let T be a term evaluation tree and let $s{=}t$ be a critical pair to be evaluated. Assume $E : (F \cup V) \times \mathbf{N} \times \mathbf{N} \times \mathbf{Z} \to \mathbf{Z}$ for the definition of $\mathtt{eval_W}$ defined by

$$E(f, count, info, k) = \begin{cases} 1 + \lfloor k/info \rfloor & \text{if } f \text{ is a variable symbol} \\ 2 + \lfloor k/info \rfloor & \text{if } f \text{ is a function symbol} \end{cases}$$

Then the value of a critical pair $s{=}t$ is defined as follows:

$$value :=$$

$$\mathtt{eval_W}(s, T, learn\_influence * TET\_size) + \mathtt{eval_W}(t, T, learn\_influence * TET\_size)$$

Note that this definition depends on *unk_penalty* and *learn_influence*. These are once again constants defined by the user for the complete proof process. Also note that the value is defined for all critical pairs, as the function E is only called for term nodes corresponding to nodes in the TET. For these nodes $info$ is guaranteed to be greater than 0.

A node is evaluated by a constant weight modified by the influence of the learned knowledge. This is expressed by a weighting factor for the learned influence multiplied by the number of examples in the TET (serving as a norming factor) divided by the sum over all terms sharing the node of the number of proofs the terms appeared in. Once more we can show that this evaluation function results is a fair strategy.

Theorem 7 : Fairness of `2tet_learn`
The strategy `2tet_learn` is fair.

Proof: Again, we only need to show that the evaluation function yields integer values greater than or equal to a base value and that only a finite number of pairs is mapped onto the same integer value.

- The functions yields an integer value by definition.
- $\mathtt{eval_W}$ sums over all the nodes in the term. All elements of this sum are positive, thus the sum is always positive and thus the value of a critical pair is always greater than 0.
- Assume that an infinite number of terms are mapped onto the same value l by the term. Then the size of the terms that are mapped onto l is not bounded, as there are only finitely many terms at a given size. But each node in the term contributes at least a weight of 1 to the sum. Thus, no term with a size greater than l is mapped onto l. This is contradictory, and thus the assumption is wrong.
  The value of a critical pair is the sum of the values of its terms. As the weight of a term is an integer greater that 0, only a finite number of pairs is mapped to the same value.

7.3 Sequential Performance of the TET-Based Strategies

As in the previous chapter we will first analyze the behaviour of the learning experts in explorative mode. Despite the high demands on computation time it was possible to conduct rather extensive tests for a variety of parameters. The results are quite satisfying.

7.3.1 Data for `1tet_learn`

The **`1tet_learn`** strategy has three parameters that can be set by the user: The penalty for term nodes not corresponding to TET nodes $unk\_penalty$, the weight given to the number of proofs terms with a certain node were used in $proofs\_w$ and the relative influence of the evaluation of the nodes found in the TET as opposed to the static weight and the weight of the nodes not found $learn\_influence$. We were able to run a complete series of tests with $proofs\_w$ equal to 1, $unk\_penalty$ ranging from 2 to 4 and $learn\_influence$ ranging from 1 to 9. Tables 7.1 and 7.2 document the results. The data is also visualized in Fig. 7.2 (corresponding to Table 7.1) and Fig 7.3 (Table 7.2).

$unk\_penalty$ / $learn\_influence$	2	3	4
1	26869.40	26742.85	25523.92
2	24132.56	25064.31	24944.68
3	23248.04	23587.35	24260.99
4	25558.21	24204.32	24701.84
5	26032.03	23577.73	23766.32
6	28035.24	25145.43	23247.64
7	29735.43	26477.33	23381.47
8	29503.80	27594.09	24797.95
9	31699.22	27677.12	27026.92

Table 7.1: Accumulated times for `1tet_learn` with $proofs_w = 1$

$unk\_penalty$ / $learn\_influence$	2	3	4
1	62	62	65
2	65	64	64
3	66	66	65
4	65	67	66
5	63	68	68
6	62	66	67
7	61	62	66
8	63	62	66
9	58	62	63

Table 7.2: Successes for `1tet_learn` with $proofs_w = 1$

It is easy to determine that the choice of the parameters plays no unimportant role for the power and behaviour of the strategy. The graphs for the different values of $unk\_penalty$ show rather similar behavior, with a pronounced minimum in the run times for values of $learn\_influence$ between 4 and 7. The minimum becomes both wider wider and smaller with increasing $unk\_penalty$, suggesting a stable trend. The minimum in the total run times

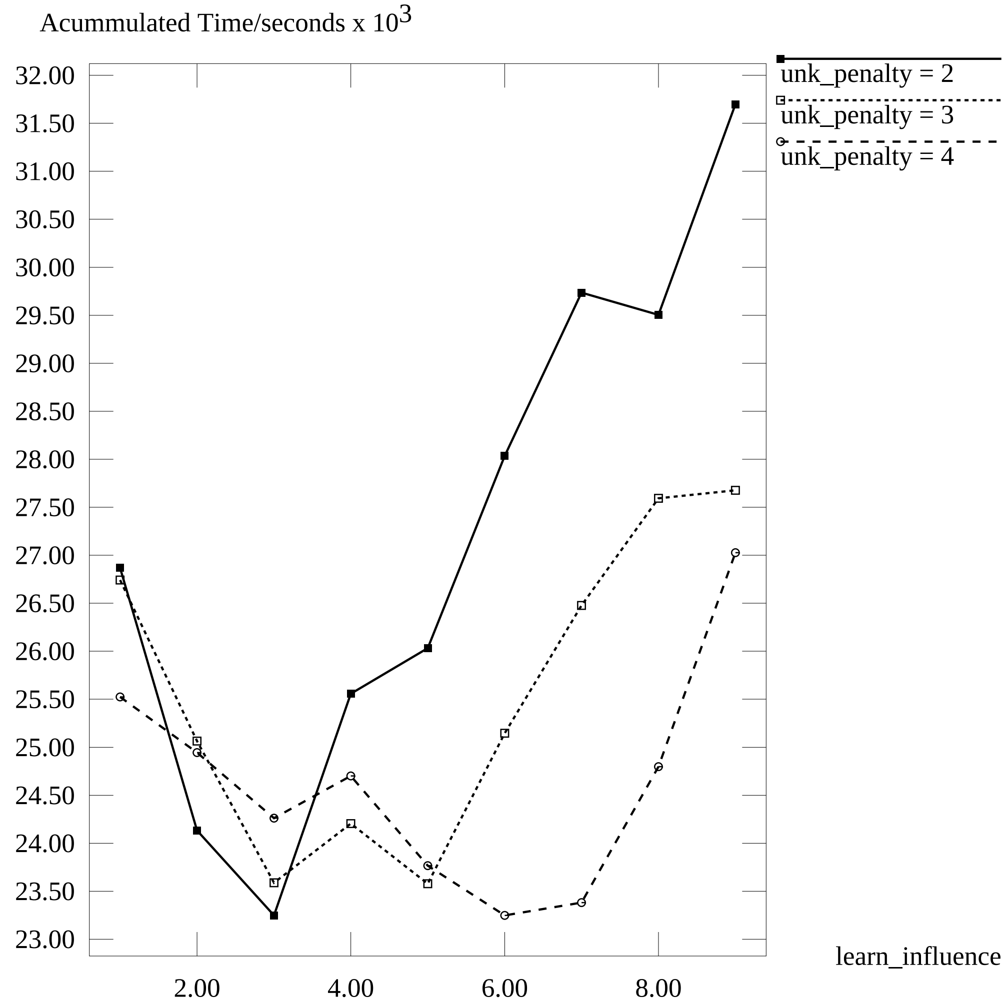


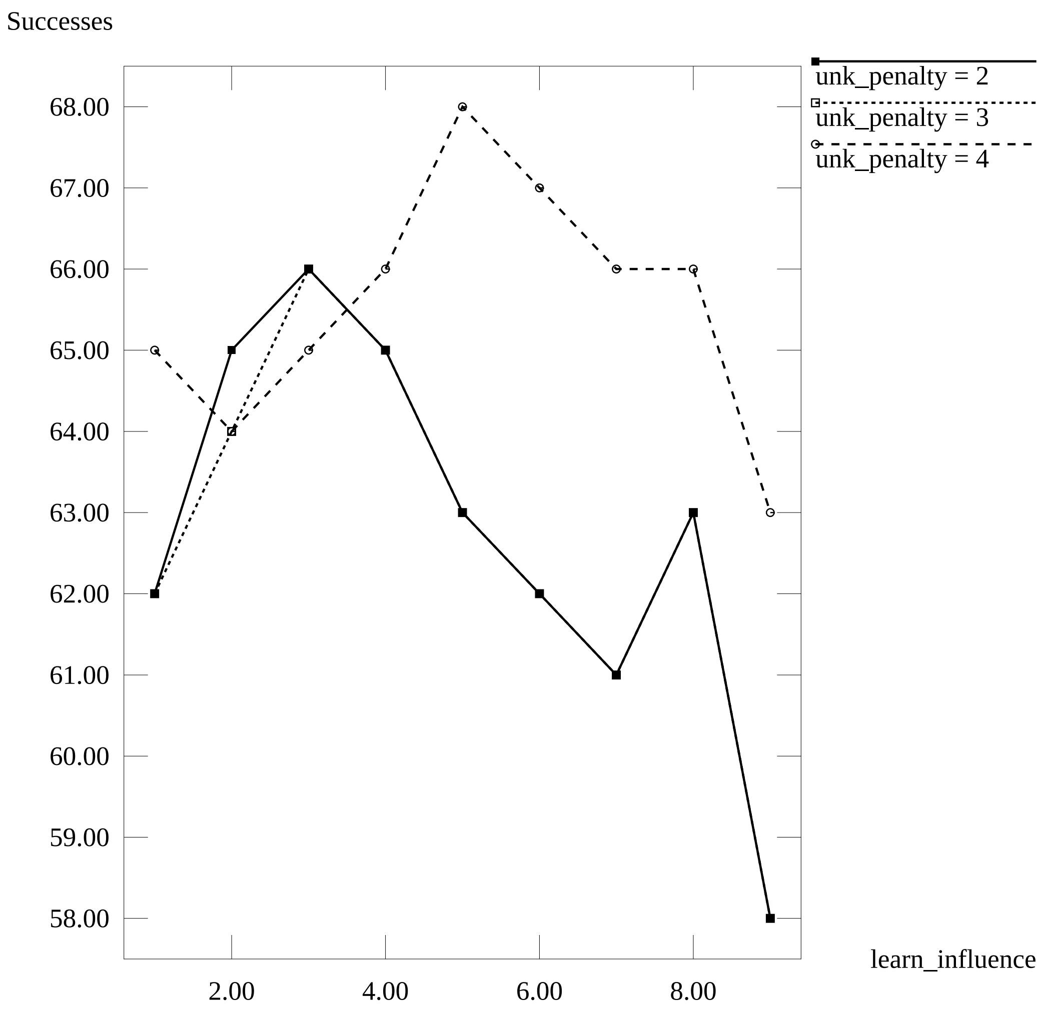
Figure 7.2: Accumulated times for `1tet_learn` with $proofs_w = 1$

is matched by a maximum in the number of proved examples. For $unk\_penalty = 4$ even the value for the previously unrivaled *occnest* expert (compare appendix A) is surpassed.

However, *occnest* and `1tet_learn` do prove different examples. The learning system has brought a number of problems into reach of the sequential prover that up to now had been confined to the team. Among these are the examples `p08b`, `p8_9a` and `p8_9b` no other expert could prove in sequential mode.

In order to test for unexpected effects we also took a couple of (semi-) random parameter combinations outside the comprehensive survey and conducted test runs on these. The results can be found in Table 7.3.

The new values are not particularly surprising. However, a general trend seems to indicate that small changes in the parameters only have small effects on the overall result. For most conventional physical systems this is to be expected. However, completion based proof



Note: For values of $learn\_influence$ above 3 the two graphs for $unk\_penalty = 2$ and $unk\_penalty = 4$ completely overlap.

Figure 7.3: Successes for `1tet_learn` with $proofs_w = 1$

systems have often shown a more erratic behaviour, were minimal changes in the parameters can have very significant effects. Two results should be commented. First, the result for $learn\_influence = 3$, $unk\_penalty = 20$, and $proofs\_w = 1$ shows that an unlimited increase in $unk\_penalty$ does not lead to a unlimited improvement. The last row but one has another interesting data point. For the values 10, 5 and 1 the fastest overall result was achieved.

We choose this parameter set for the persistent and the reproductive modes. The results can be found in Table 7.4. We were rather surprised about the poor performance of the strategy in reproductive mode – it performs slightly worse than *add*. We suppose that the TET based strategies cannot cope with to large knowledge bases very well. The results in the third column of the table seem to support this view. In this we used only the lemmas from

learn_influence	*unk_penalty*	*proofs_w*	**Accumulated time**	**Successes**
2	1	0	28188.42	61
3	1	0	27225.58	61
3	2	2	24010.74	66
3	20	1	40237.02	48
3	3	2	25805.97	66
3	4	2	23692.04	67
3	5	2	26335.09	63
3	6	2	27252.58	62
3	7	2	28817.82	63
3	8	2	30822.94	60
3	9	2	32447.31	58
4	1	0	27262.57	61
5	5	1	23456.43	68
5	5	2	23777.02	67
5	5	3	23984.97	66
5	10	1	26706.31	63
6	10	2	27178.49	65
6	5	1	23122.11	68
6	5	2	23443.43	67
10	5	1	21833.33	68
10	5	2	24454.94	67

Table 7.3: Results for the strategy `1tet_learn` with assorted parameter combinations

the complete knowledge base to build the TET[5]. The results warrant further investigation in the future. The problem might be overcome by the use of some kind of knowledge evolution mechanism including forgetting, or by the implementation of TET based strategies using only a part of the knowledge base (as the goal and specification bound experts from the previous chapter).

	Persistent	**Reproductive**	**Reproductive (Lemmas only)**
Successes	68	57	64
Accumulated Time	22604.16	31563.67	23932.72

Note: The parameters $learn\_influence = 10$, $unk\_penalty = 5$, and $proofs\_w = 1$ were used for these results.

Table 7.4: `1tet_learn` in persistent and reproductive mode

Concluding this chapter, we present detailed data on the performance of the strategy on the single examples. The tables 7.5 to 7.7 give the run times for the two reference strategies *add*

[5] As this is not yet supported by the current implementation we had to change the source and recompile the system. For this reasons no more extensive tests were conducted.

and *occnest*, followed by the times for `1tet_learn` with the parameters $learn\_influence = 10$, $unk\_penalty = 5$, and $proofs\_w = 1$. Four different modes are presented: Explorative, persistent, reproductive and reproductive (lemmas only).

It is interesting to compare the behaviour of *occnest* with the behaviour of `1tet_learn`. In most cases *occnest* finds a proof, he needs very short time. There is no proof *occnest* needs more than 200 seconds for, and only examples take more than 30 seconds. The times for the learning expert cover a much wider spread. This can be interpreted as an indicator for one of *occnest*'s disadvantages. It uses a very specialized heuristic tailored to certain problems and breaks down for other cases. `1tet_learn`, on the other hand, uses a very global concept and consequently is much harder to bring completely of track.

Fig 7.4 presents the information about the run times in an easier to grasp graphical form. The figure shows accumulated run time versus the number of examples tried. The results for reproductive mode using lemmas only has been ommited for clarity. It would fall in slightly above the graph for persistent mode.

As can be seen, the new, learning strategy is quite competitive in explorative mode, outperforming *add* and, for quite some examples even beating *occnest*. The reproductive mode, on the other hand, is not very successful. However, we can see that in reproductive mode the learning strategy performs very bad for the simple examples at the beginning. It performs similar to the other strategies for the later examples but cannot regain the lost time. This again indicates that to much knowledge might be a burden for the strategy. We will keep this in mind for the tests with the team incorporating learning experts.

7.3.2 Data for `2tet_learn`

The `2tet_learn` strategy only uses two parameters: $unk\_penalty$ and $learn\_influence$. We have conducted more than 50 complete test with this strategy in explorative mode. This series alone used more than 500 hours of raw cpu time on various SUN ELC workstations. All possible combinations with $unk\_penalty$ between 1 and 4 and $learn\_influence$ ranging from 1 to 10 have been conducted. The tables 7.8 and 7.9 show the cumulated run times and the number of successes, respectively. Figures 7.5 and 7.6 make the general trend more obvious.

First, the behaviour of the strategy is much more erratical than that of `1tet_learn`. However, one observation can still be made. There is a strong tendency for examples with a larger value of $learn\_influence$ to be perform better. This can be seen in both figures, although the it is more visible in the first one. To further investigate this phenomenon, we performed additional tests for large values of $learn\_influence$. They are collected in table 7.10.

For brevities sake we refrain from printing yet another table with complete results. All in all the `2tet_strategy` performs rather similar to the first TET based strategy.

Examples	add	occnest	1tet_learn			
			Explor.	Persist.	Repro.	Lemmas
ax_antisyma	0.230	0.190	0.280	0.320	0.700	0.490
ax_antisymb	0.210	0.230	0.230	0.320	0.670	0.340
ax_glb1a	27.790	0.820	15.000	14.210	–	77.630
ax_glb1b	0.730	0.290	0.430	0.480	0.690	0.380
ax_glb1c	28.110	0.820	3.480	3.450	11.960	11.240
ax_glb1d	0.800	1.020	0.420	0.550	–	49.280
ax_glb2a	0.220	0.200	0.210	0.310	0.620	0.360
ax_glb2b	0.250	0.210	0.260	0.320	0.670	0.390
ax_glb3a	0.220	0.210	1.090	1.090	–	6.660
ax_glb3b	0.230	0.210	0.280	0.310	0.650	0.400
ax_lub1a	2.490	0.270	0.500	0.560	0.910	0.580
ax_lub1b	42.900	2.160	3.720	3.630	–	36.500
ax_lub1c	42.750	2.200	3.610	3.490	3.250	3.120
ax_lub1d	2.470	1.050	1.720	1.770	–	15.910
ax_lub2a	0.290	0.220	0.350	0.390	0.700	0.440
ax_lub2b	0.200	0.220	0.270	0.320	0.620	0.340
ax_lub3a	0.260	0.230	0.310	0.370	0.670	0.420
ax_lub3b	0.210	0.210	0.290	0.320	0.660	0.380
ax_mono1a	46.670	0.200	4.040	3.950	6.050	5.930
ax_mono1b	47.920	0.220	4.070	4.010	4.360	4.050
ax_mono1c	47.560	0.300	2.090	4.050	6.340	6.050
ax_mono2a	44.860	0.220	1.800	1.850	3.960	3.350
ax_mono2b	46.380	0.220	1.260	1.310	2.770	2.330
ax_mono2c	45.120	0.240	3.250	3.230	–	20.960
ax_refla	0.210	0.190	0.290	0.310	0.660	0.350
ax_reflb	0.160	0.170	0.250	0.310	0.630	0.360
ax_transa	0.460	1.450	0.580	0.630	0.890	0.650
ax_transb	0.430	1.450	0.300	0.410	0.680	0.430
distrnu	–	–	–	–	–	–
distrun	–	–	–	–	–	–
lat1a	6.550	0.270	2.070	2.050	3.960	3.280
lat1b	6.570	0.210	2.130	2.100	4.170	3.340
lat2a	9.310	24.160	17.840	17.390	6.010	4.830
lat2b	9.160	21.46	2.580	2.590	6.150	5.100
lat3a	56.670	0.250	2.900	2.870	23.560	18.070
lat3b	56.960	0.260	4.630	4.610	33.600	23.100
lat4	893.50	–	291.950	270.090	–	228.360

Table 7.5: Comparative performance of 1tet_learn

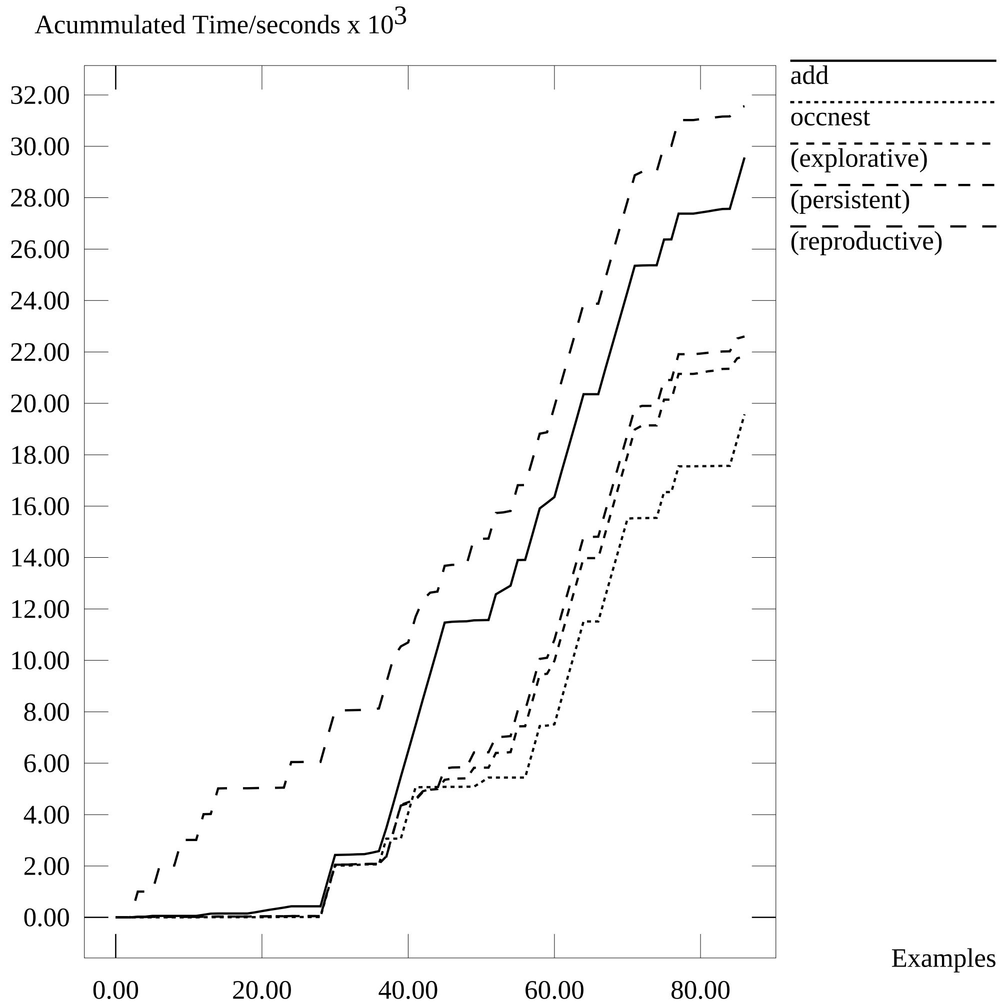
Examples	add	occnest	1tet_learn			
			Explor.	**Persist.**	**Repro.**	**Lemmas**
p01a	–	0.480	–	–	–	–
p01b	–	0.460	–	–	416.310	–
p02a	–	–	110.310	83.730	153.940	–
p02b	–	–	87.510	167.860	–	–
p03a	–	4.400	304.940	300.110	652.460	129.390
p03b	–	2.690	90.910	89.640	286.550	127.600
p03c	–	2.760	24.350	24.600	43.310	27.640
p03d	–	4.370	361.710	766.810	–	99.860
p04a	34.220	2.000	44.900	47.940	37.340	53.050
p04b	9.700	1.930	3.130	3.040	3.290	2.700
p04c	9.790	1.900	5.990	5.710	6.070	5.070
p04d	33.780	2.020	411.310	573.270	–	71.280
p05a	7.930	177.35	4.180	4.120	4.320	3.540
p05b	7.960	177.92	4.160	4.060	4.350	3.620
p06a	–	0.630	568.600	593.390	–	–
p06b	165.810	0.320	8.610	8.070	21.610	11.850
p06c	167.190	0.310	26.740	28.410	59.710	23.100
p06d	–	0.570	–	–	–	–
p07	5.100	0.190	3.460	3.380	3.540	3.310
p08a	–	–	–	–	–	–
p08b	–	–	–	–	–	–
p09a	219.480	20.470	42.250	43.420	53.840	32.870
p09b	224.350	52.330	502.100	705.320	–	141.570
p10	–	–	–	–	–	–
p11	–	–	–	–	–	–
p12	–	–	–	–	–	–
p12x	–	–	–	–	–	–
p17a	0.280	0.200	0.470	0.670	0.730	0.510
p17b	0.250	0.210	0.430	0.560	0.620	0.380
p18	–	–	–	–	–	–
p19	–	–	–	–	–	–
p20	–	–	–	–	–	–
p20x	–	–	–	–	–	–
p21	–	19.520	–	–	–	–
p21x	7.890	0.300	156.500	90.470	135.820	3.400

Table 7.6: Comparative performance of 1tet_learn (continued)

Examples	add	occnest	1tet_learn			
			Explor.	**Persist.**	**Repro.**	**Lemmas**
p22a	5.100	0.970	3.540	3.580	3.690	3.250
p22b	5.040	11.070	3.490	3.430	3.670	3.240
p23	–	–	–	–	–	–
p23x	4.770	0.260	2.080	2.110	2.230	1.890
p33	–	–	–	–	–	–
p38a	0.250	0.190	0.560	0.700	0.670	0.440
p38b	0.230	0.180	0.470	0.620	0.670	0.410
p39a	45.210	5.460	33.620	26.170	34.020	6.020
p39b	45.400	4.070	62.910	29.860	35.170	6.200
p39c	45.340	4.040	35.190	26.290	34.220	6.100
p39d	45.250	5.460	60.860	28.040	35.020	6.180
p40a	5.240	2.030	2.320	2.390	2.450	2.150
p8_9a	–	–	408.190	499.370	314.780	–
p8_9b	–	–	83.060	83.050	86.710	620.700
Successes	59	67	68	68	57	64
Failures	27	19	18	18	29	22
Accumulated Time	29564.4	19569.1	21833.33	22604.16	31563.67	23932.72

Note: The `1tet_learn` strategy used the parameters $learn\_influence = 10$, $unk\_penalty = 5$, and $proofs\_w = 1$. *occnest* and *add* are reproduced from appendix A for easy comparisons. All times are in seconds. A dash indicates that no proof was found within the 1000 seconds time limit.

Table 7.7: Comparative performance of `1tet_learn` (concluded)



Note: The last three entries in the legend refer to `1tet_learn` runs in the described modes.

Figure 7.4: Comparative performance of `1tet_learn`

$unk\_penalty$ / $learn\_influence$	1	2	3	4
1	29565.54	26739.95	28072.15	27169.45
2	26763.85	26552.11	26668.40	27156.97
3	28036.60	26738.21	26517.96	26979.24
4	24645.22	25772.93	25965.88	26035.65
5	25234.67	25970.68	25685.65	25712.43
6	26416.47	27386.86	27204.14	27083.99
7	25100.44	26017.03	25915.04	26220.24
8	25463.63	26158.39	26030.95	25972.43
9	24910.65	25582.93	25576.57	25540.40
10	25702.37	25782.10	25703.60	25948.14

Table 7.8: Accumulated times for `2tet_learn`

$unk\_penalty$ / $learn\_influence$	1	2	3	4
1	59	63	61	61
2	63	62	62	61
3	61	62	62	62
4	63	65	65	65
5	65	65	65	65
6	62	65	65	65
7	65	65	65	65
8	65	65	65	65
9	65	65	65	65
10	64	64	64	64

Table 7.9: Number of successful proofs for `2tet_learn`

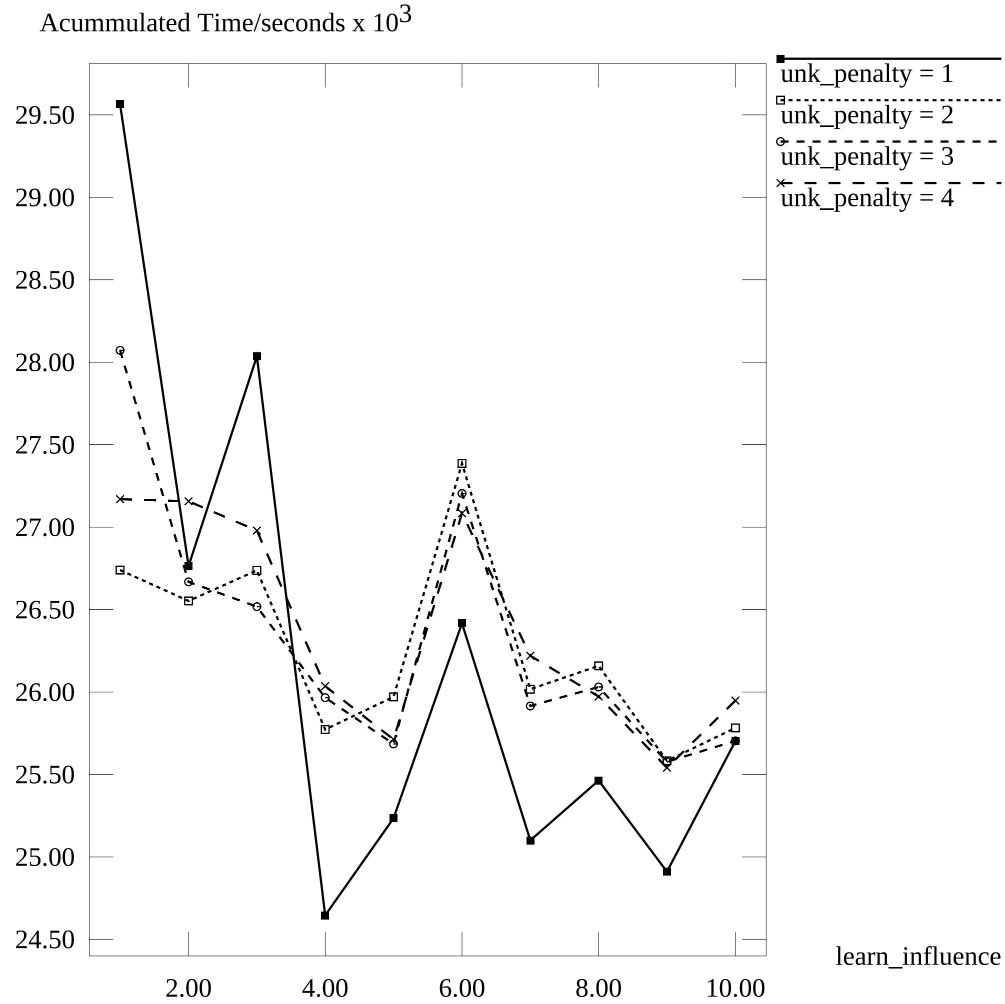


Figure 7.5: Accumulated times for `2tet_learn`

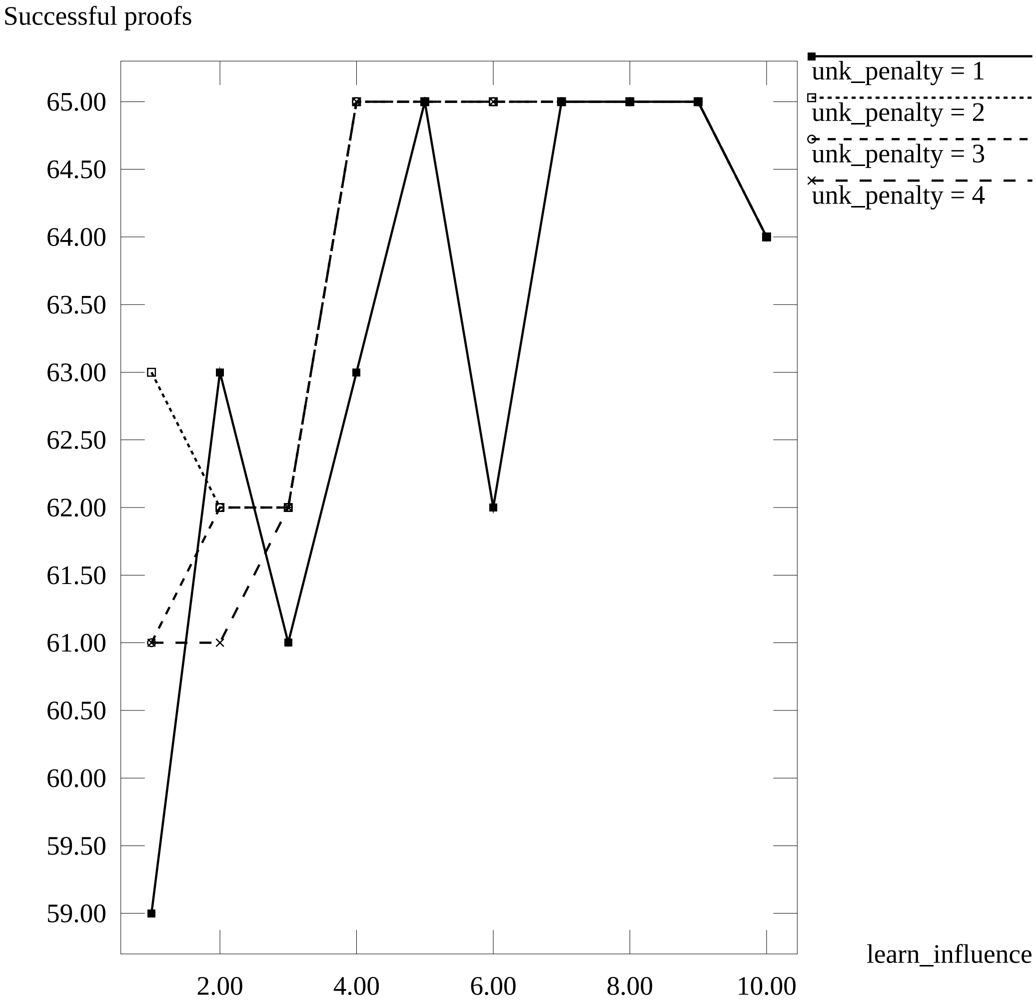


Figure 7.6: Successful proofs for `2tet_learn`

$learn\_influence$	$unk\_penalty$	**Successes**	**Accumulated time**
10	1	64	25702.37
15	1	66	22906.81
20	1	67	23248.76
30	1	67	24014.12
40	1	66	24251.11
10	2	64	25782.10
30	2	67	24213.39
10	3	64	25703.60
20	3	67	23201.93
40	3	66	24532.80
50	3	66	24669.52
10	4	66	24669.52
20	4	67	23301.91
20	5	67	23654.42
40	5	66	24444.36
50	5	66	24404.24

Table 7.10: `2tet_learn` for assorted parameters

Chapter 8

Learning Experts in the Team

As the previous chapters showed the learning experts are powerful alternatives to the conventional ones. However, they are not the ultimate solution in the field of proof heuristics. In order to further increase their power we integrate the learning experts into a TEAMWORK team. As DISCOUNT's experts are little more than evaluation strategies, this is easy to accomplish. DISCOUNT's expert parser has been extended to recognize the learning experts, and expert descriptions for them have been created. See section D.2 for details.

8.1 Evaluation Method

Previous sections have shown that a very large knowledge base is not necessarily an advantage for all the proof strategies. Moreover, it is not reasonable to run the complete team on trivial examples. We did therefore split the set of examples into 2 parts – training examples and test examples. The *training examples* are all examples that can be proved by a conventional strategy in less then 4 seconds. In four seconds the distributed version of DISCOUNT has started and is comfortably running. Thus, a fair comparison is possible. The training examples were inserted into DISCOUNT's knowledge base and the system was then run on the remaining examples[1].

The classification led to 53 training examples: `ax_antisyma`, `ax_glb3a`, `ax_lub3a`, `ax_refla`, `p01a`, `p06a`, `p22a`, `ax_antisymb`, `ax_glb3b`, `ax_lub3b`, `ax_reflb`, `p01b`, `p06b`, `p23x`, `ax_glb1a`, `ax_lub1a`, `ax_mono1a`, `ax_transa`, `p03b`, `p06c`, `p38a`, `ax_glb1b`, `ax_lub1b`, `ax_mono1b`, `ax_transb`, `p03c`, `p06d`, `p38b`, `ax_glb1c`, `ax_lub1c`, `ax_mono1c`, `lat1a`, `p04a`, `p07`, `p40a`, `ax_glb1d`, `ax_lub1d`, `ax_mono2a`, `lat1b`, `p04b`, `p17a`, `ax_glb2a`, `ax_lub2a`, `ax_mono2b`, `lat3a`, `p04c`, `p17b`, `ax_glb2b`, `ax_lub2b`, `ax_mono2c`, `lat3b`, `p04d`, `p21x`.

23 examples remained as test cases: `lat2a`, `p02b`, `p05b`, `p10`, `p22b`, `p39d`, `lat2b`, `p03a`, `p08b`, `p11`, `p39a`, `p8_9a`, `lat4`, `p03d`, `p09a`, `p18`, `p39b`, `p8_9b`, `p02a`, `p05a`, `p09b`, `p21`, `p39c`.

They were given to both the latest standard team for DISCOUNT (including an improved supervisor with powerful planning component – compare [DK94]), and to a team containing predominantly learning experts.

The standard team contained a number of powerful and proven experts. It started with a

[1] Due to a technical error only examples for which at least one proof existed at all were used as test problems. The error was detected to late to be corrected.

standard expert and the planning expert. The team with the learning experts (called *learning team* for brevity) contained the same experts. However, it also contained 3 learning experts and started out with using only them. The following configuration file describes the learning team:

```
HOSTS          christie

DATA_DIR       /usr/users/stschulz/DISCOUNT_VERSIONS/DISCOUNT2.0
EXPERT_DIR     /usr/users/stschulz/DISCOUNT_VERSIONS/DISCOUNT2.0/EXPERT
REFEREE_DIR    /usr/users/stschulz/DISCOUNT_VERSIONS/DISCOUNT2.0/REFEREE
DOMAIN_DIR     /usr/users/stschulz/DISCOUNT_VERSIONS/DISCOUNT2.0/DOMAIN
KNOWLEDGE_DIR  /usr/users/stschulz/DISCOUNT_VERSIONS/DISCOUNT2.0/KNOWLEDGE


INITIAL      (C_1TET_LEARN (0) STAT_LAST_1)

             (C_GLOBAL_LEARN (0) STAT_LAST_1)

SELECT       (DATABASE (0) STAT_STAT_1 )

             (C_2TET_LEARN (0) STAT_LAST_1)

             (ADD_RWEIGHT (0) STAT_LAST_1)

             (GOALMATCH (0) STAT_LAST_1)

             (GOALSIM (0) STAT_LAST_1)

             (MAX_WEIGHT (0) STAT_LAST_1)

             (LATTICE_1 (0) STAT_STAT_1)

             (LATTICE_2 (0) STAT_STAT_1_NO_M_LATT)

             (OCCNEST (0) STAT_STAT_1)

DOMAIN_SELECT   LATTICE


SUPERVISOR_FILE
   /usr/users/stschulz/DISCOUNT_VERSIONS/DISCOUNT2.0/PLAN/std_plan.CFG
```

8.2 Results and Evaluation

Table 8.1 contains the results of the test runs. The time for each single example was restricted to 500 seconds in this case. The learning team outperforms the standard team in both the number of examples proved and in the run times for nearly any example. This indicates that the learning experts cooperate very well both with each other and with the standard experts.

Example	**Standard team**	**Learning team**
lat2a	25.7	8.6
lat2b	33.3	9.1
lat4	–	81.1
p02a	–	–
p02b	–	–
p03a	9.9	9.2
p03d	9.8	8.0
p05a	39.2	7.6
p05b	39.7	7.8
p08b	–	97.1
p09a	55.8	31.0
p09b	57.9	30.6
p10	–	–
p11	–	–
p18	–	–
p21	–	–
p22b	20.6	6.6
p39a	25.3	15.9
p39b	17.3	15.5
p39c	17.6	15.4
p39d	26.0	15.7
p8_9a	–	–
p8_9b	7.7	–

Table 8.1: Runtimes for the standard team and the learning team

Additional data has to be collected before a final verdict can be given. However, it seems likely that learning experts can be a valuable addition to almost any team.

Chapter 9

Conclusion

This work is far from the ultimate answer to all questions about machine learning and equational deduction. On the contrary, as one of the first works on this field it opens a wide door to new problems. However, a number of facts can be stated now.

- Old proofs definitely carry information useful for the guidance of new problems. Considering positive examples only is sufficient to achieve significant improvements.
- The knowledge for the guidance of proofs can be stated in a very compact way. Small knowledge bases and symbolic learning algorithms can successfully be used to tackle proof problems.
- The power of DISCOUNT has been significantly improved for both the sequential and the distributed mode. Learning strategies can be combined with conventional ones very successful.

The main aim of this work has bee achieved – new, learning experts boost the power of DISCOUNT. However, a large amount of work is still to be done.

- Further experiments are necessary to improve the TET based strategies handling of large knowledge bases. A first step could consist in testing the strategies with varying amount of knowledge. However, in the end an automatic mechanism for the selection of optimal knowledge will be desired.
- A large variety of generalization schemes can be imagined to improve the extrapolative power of the reuse-based strategies. For example, the use of abstraction trees for the indexing of knowledge can be used to integrate first order generalizations.
- The behaviour of the learning experts in the team has to be studied further to obtain more data and experiences. Learning experts have to be integrated into the supervisors planning attempts.
- Finally, DISCOUNT's growth probably has reached its limit. Originally intended as a prototypical prover for the demonstration of TEAMWORK, it has by now been extended in a variety of ways. Up to now it scaled remarkably well, however, each new addition introduces new error sources. To save the large body of experiences collected

with the system, a new prover with an architecture more consequently designed for the current and planned extensions would be extremely useful.

Appendix A

The Domain of Lattice Ordered Groups

Problems for (equational) theorem provers are usually generated in a rather *ad hoc* fashion. A rather large number of small, unrelated problem classes do exist, but there is a distinct lack of problems scalable to various difficulties, or of a domain with a variety of problems covering a wide range of difficulties. This lack has made the unbiased evaluation of equational proof systems rather difficult.

However, we are now in a more fortunate position. The DISCOUNT system was, among other provers, used by the group of Ingo Dahn at the Humbold University in Berlin for the construction of a proof pad for mathematicians (the ILF project – compare [Da+94]).

ILF was used on the test case of lattice ordered group. In the course of the evaluation a large number of problems from this domain have been generated and given to DISCOUNT. We have, for some time now, used these examples from a field of active interest to mathematicians in the evaluation of DISCOUNT. In the course of this work a comprehensive survey on a large subclass of the problems has been conducted and is used in the evaluation of the new learning component of DISCOUNT.

A.1 Lattice Ordered Groups – an Equational Axiomatization

This section gives the definition of lattice ordered groups and suggests a set of equational axioms whose models are the lattice ordered groups. We start with the well-known definitions of a group and a partially ordered set.

Definition 16 : Groups
Let M be a set of elements, and let $f : M \times M \to M$ be a binary operator on M. (M, f) is called a group, if the following axioms hold:

1. f is associative:
   $\forall x, y, z \in M : f(x, f(y, z)) = f(f(x, y), z)$

2. There exists a neutral element with respect to f:
   $\exists 1 \in M \forall x \in M : f(1, x) = x$

3. For each x there exists an inverse element with respect to f:
   $\forall x \in M \exists y \in M : f(y, x) = 1$

Definition 17 : Partially ordered set
Let M be a set of elements and let $\leq \subseteq (M \times M)$ be a binary relation over M. $(M, \leq)$ is a partially ordered set, if $\leq$ has the following properties:

1. $\leq$ is reflexive: $\forall x \in M : x \leq x$

2. $\leq$ is anti-reflexive: $\forall x, y \in M : x \leq y, y \leq x \rightsquigarrow x = y$

3. $\leq$ is transitive: $\forall x, y, z \in M : x \leq y, y \leq z \rightsquigarrow x \leq z$

$\leq$ is called a *partial ordering.*

A *lattice* is a partially ordered set with greatest lower and least upper bounds:

Definition 18 : Lattice
Let $(M, \leq)$ be a partially ordered set. $(M, \leq)$ is called a lattice, if the following holds:

1. For each pair x, y there exists a greatest lower bound:
   $\forall x, y \in M \exists n \in M \forall z \in M : n \leq x, n \leq y, z \leq x, z \leq y \rightsquigarrow z \leq n$

2. For each pair x, y there exists a least upper bound:
   $\forall x, y \in M \exists u \in M \forall z \in M : x \leq u, y \leq u, x \leq z, y \leq z \rightsquigarrow u \leq z$

Finally, a *lattice ordered group* is a structure that is a lattice with respect to $\leq$, a group with respect to f and that fulfills certain monotony laws.

Definition 19 : Lattice ordered Group
Let M be a set of elements, $f : M \times M \to M$ be a binary operator on M and $\leq \subseteq (M \times M)$ be a binary relation over M. $(M, f, \leq)$ is a lattice ordered group if the following holds:

1. (M, f) is a group.

2. $(M, \leq)$ is a lattice.

3. f is monotonic with respect to $\leq$:

   - $\forall x, y, z \in M : x \leq y \rightsquigarrow f(x, z) \leq f(y, z)$
   - $\forall x, y, z \in M : x \leq y \rightsquigarrow f(z, x) \leq f(z, y)$

In [Fu94] the author proves that the axioms can be restated in a purely equational calculus. The group is axiomatized as usual (see for example [KB70]), the partial ordering is recoded in properties of the upper and lower bound. The following theorem holds:

Theorem 8 : Axiomatization of lattice ordered groups
The following specification describes the lattice ordered groups – each model of the axioms is a lattice ordered group and for each lattice ordered group the axioms hold[1].

Group Axioms

```
f(x,f(y,z)) = f(f(x,y),z)
f(1,x) = x
f(i(x),x) = 1
```

Specification of the least upper bound `u` *and the greatest lower bound* `n`

```
n(x,y) = n(y,x)
u(x,y) = u(y,x)
n(x,n(y,z)) = n(n(x,y),z)
u(x,u(y,z)) = u(u(x,y),z)
u(x,x) = x
n(x,x) = x
u(x,n(x,y)) = x
n(x,u(x,y)) = x
```

Monotony of `f`

```
f(x,u(y,z)) = u(f(x,y),f(x,z))
f(x,n(y,z)) = n(f(x,y),f(x,z))
f(u(y,z),x) = u(f(y,x),f(z,x))
f(n(y,z),x) = n(f(y,x),f(z,x))
```

We will refer to this set of axioms as `LOG`.

The proof for the soundness of the axiomatization makes use of the uniqueness of the upper and lower bound for each pair. The proof of completeness has been splitted into separate proofs, each one demonstrating that one of the original axioms is implied by the equational axiomatisation. These proofs have been conducted by DISCOUNT, the corresponding problems form a part of the problem set. All problems with names that start in "`ax_`" are derived by proving one of the original axioms of lattice ordered groups in the axiomatization from theorem 8.

A variety of problems have conditional goals. These are transformed into simple equational problems by skolemization and application of modus ponens. The typical specification for a problem in the domain thus consists of the axioms in `LOG`, optionally some additional ground equations (skolemized conditions from the conditional goal) and a skolemized (ground) equation, the conclusion to be shown. Quite often some implications of the `LOG` axioms are added as lemmas to make the problem in question less difficult.

A.2 Description of the Examples

This section provides a short description of the problems. For this work a problem set consisting of 86 different examples has been selected. More examples from the domain are

[1]As usual with equational specifications the equations are assumed to be universally quantified.

generated continuously – the selected set is only a snapshot of the established examples at a given time.

As we saw above, the partial ordering $\leq$ is being expressed in terms of the upper and lower bounds in the axiomatization. The predicate $s \leq t$ can be expressed in two different forms. One possibility uses the least upper bound, yielding $\mathtt{u(s,t)} = \mathtt{t}$. The other one uses the greatest lower bound, resulting in the clause $\mathtt{n(s,t)} = \mathtt{s}$. In conditional goals the predicate $\leq$ can appear more than once and the recoding can result in one of many possible forms. We adopt the convention that examples differing only in this encoding of $\leq$ use the same base name with a different suffix. Thus the problem `ax_antisyma` encodes the $\leq$ predicate using `u` and the problem `ax_antisymb` uses `n`. Tables A.1 to A.4 contain a short description for all examples.

A.3 Performance of the Conventional Experts

We stated above that the examples presented here are of widely varying difficulty. To back up this claim and to provide some valid reference points for the evaluation of the learning experts, we have included data on the performance of the conventional (forward- and goal oriented) experts. Tables A.5 to A.7 contain the run times of the DISCOUNT system using different heuristics. The times recorded are seconds of cpu time on a SUN ELC. They correspond very closely to the real time on an otherwise unused machine with sufficient main memory to avoid serious swapping.

As with all our tests the (cpu) time was restricted to 1000 seconds per problem to keep the complete time for a test run down to an acceptable maximum. The accumulated time at the end of the last table assumes 1000 seconds for examples no proof was found within the time limit. This allows at least a qualitative comparison of all different strategies.

The *occnest*, *fifo* and *add* strategies have already been explained in chapter 4. The *max* and the *gt* strategies are related to the *add* strategies. The *max* strategy prefers critical pairs were the maximum of the weights of the two terms is small. *gt* evaluates only the weight of the term that is, in the reduction ordering, greater. It uses the sum of the weights if the two terms are incomparable in this ordering. Again, small weights are prefered. Test runs for the similarity-based goal directed experts have not been included. These experts did not perform up to any reasonable standard in sequential mode[2].

The data from the table is visualized in Fig. A.1. It shows the accumulated run time versus the number of examples handled. The examples were given to DISCOUNT in the order they appear in the tables. Again, 1000 seconds have been assumed for the accumulated time if no proof was found. It can be seen that *occnest* is indeed very well adapted to the domain, outclassing all other strategies. The all-purpose *add* strategy also performs acceptably. The *fifo* strategy is, as usual, not competitive. An interesting and unexpected fact is, however, that even *fifo* outclasses the *gt* strategy. *gt* performs unusually bad in this domain and is only able to prove 5 of the 86 examples.

[2]They can be very valuable in the team, though. Compare [DF94] and [Fu94].

Name	Axioms	Goal
ax_antisyma	LOG ∪ {u(a,b) = b, u(a,b) = a}	a = b
ax_antisymb	LOG ∪ {n(a,b) = b, n(a,b) = a}	a = b
ax_glb1a	LOG ∪ {u(a,c) = a, u(b,c) = b}	u(n(a,b),c) = n(a,b)
ax_glb1b	LOG ∪ {n(a,c) = c, n(b,c) = c}	n(n(a,b),c) = c
ax_glb1c	LOG ∪ {n(a,c) = c, n(b,c) = c}	u(n(a,b),c) = n(a,b)
ax_glb1d	LOG ∪ {u(a,c) = a, u(b,c) = b}	n(n(a,b),c) = c
ax_glb2a	LOG	u(n(a,b),a) = a
ax_glb2b	LOG	n(n(a,b),a) = n(a,b)
ax_glb3a	LOG	u(n(a,b),b) = b
ax_glb3b	LOG	n(n(a,b),b) = n(a,b)
ax_lub1a	LOG ∪ {u(a,c) = c, u(b,c) = c}	u(u(a,b),c) = c
ax_lub1b	LOG ∪ {n(a,c) = a, n(b,c) = b}	n(u(a,b),c) = u(a,b)
ax_lub1c	LOG ∪ {u(a,c) = c, u(b,c) = c}	n(u(a,b),c) = u(a,b)
ax_lub1d	LOG ∪ {n(a,c) = a, n(b,c) = b}	u(u(a,b),c) = c
ax_lub2a	LOG	u(a,u(a,b)) = u(a,b)
ax_lub2b	LOG	n(a,u(a,b)) = a
ax_lub3a	LOG	u(b,u(a,b)) = u(a,b)
ax_lub3b	LOG	n(b,u(a,b)) = b
ax_mono1a	LOG ∪ {u(a,b) = b}	u(f(a,c),f(b,c)) = f(b,c)
ax_mono1b	LOG ∪ {n(a,b) = a}	n(f(a,c),f(b,c)) = f(a,c)
ax_mono1c	LOG ∪ {u(a,b) = b}	n(f(a,c),f(b,c)) = f(a,c)
ax_mono2a	LOG ∪ {u(a,b) = b}	u(f(c,a),f(c,b)) = f(c,b)
ax_mono2b	LOG ∪ {n(a,b) = a}	n(f(c,a),f(c,b)) = f(c,a)
ax_mono2c	LOG ∪ {n(a,b) = a}	u(f(c,a),f(c,b)) = f(c,b)
ax_refla	LOG	u(a,a) = a
ax_reflb	LOG	n(a,a) = a
ax_transa	LOG ∪ {u(a,b) = b, u(b,c) = c}	u(a,c) = c
ax_transb	LOG ∪ {n(a,b) = a, n(b,c) = b}	n(a,c) = a

Note: In this and the following tables (as well as in the the actual specification files for DISCOUNT) a, b, c ... denote (skolem) constants, while x, y, z, ... denote variables.

Table A.1: Problems in the domain of lattice ordered groups

Name	Axioms	Goal
distrnu	LOG	u(a,n(b,c)) = n(u(a,b),u(a,c))
distrun	LOG	n(a,u(b,c)) = u(n(a,b),n(a,c))
lat1a	LOG ∪ {u(a,1) = a}	u(a,f(a,a)) = f(a,a)
lat1b	LOG ∪ {n(a,1) = a}	n(a,f(a,a)) = f(a,a)
lat2a	LOG ∪ {u(a,1) = a, u(b,1) = b}	u(a,f(a,b)) = f(a,b)
lat2b	LOG ∪ {n(a,1) = a, n(b,1) = b}	n(a,f(a,b)) = a
lat3a	LOG ∪ {u(a,1) = a, u(b,1) = b}	u(a,f(b,a)) = f(b,a)
lat3b	LOG ∪ {n(a,1) = a, n(b,1) = b}	n(a,f(b,a)) = a
lat4	LOG ∪ {i(i(x)) = x, i(1) = 1, i(f(x,y)) = f(i(y),i(x)), pp(x) = u(x,1), np(x) = n(x,1), u(x,n(y,z)) = n(u(x,y),u(x,z)), n(x,u(y,z)) = u(n(x,y),n(x,z))}	a = f(pp(a),np(a))
p01a	LOG ∪ {u(a,b) = b}	u(f(i(c),f(a,c)),f(i(c), f(b,c))) = f(i(c),f(b,c))
p01b	LOG ∪ {n(a,b) = a}	n(f(i(c),f(a,c)),f(i(c), f(b,c))) = f(i(c),f(a,c))
p02a	LOG ∪ {u(i(a),i(b)) = i(b)}	u(a,b) = a
p02b	LOG ∪ {n(i(a),i(b)) = i(a)}	n(a,b) = b
p03a	LOG ∪ {u(a,b) = b, u(c,d) = d}	u(f(a,c),f(b,d)) = f(b,d)
p03b	LOG ∪ {n(a,b) = a, n(c,d) = c}	n(f(a,c),f(b,d)) = f(a,c)
p03c	LOG ∪ {u(a,b) = b, u(c,d) = d}	n(f(a,c),f(b,d)) = f(a,c)
p03d	LOG ∪ {u(a,b) = b, n(c,d) = c}	u(f(a,c),f(b,d)) = f(b,d)
p04a	LOG ∪ {u(1,a) = a, u(1,b) = b}	u(1,f(a,b)) = f(a,b)
p04b	LOG ∪ {n(1,a) = a, n(1,b) = b}	n(1,f(a,b)) = f(a,b)
p04c	LOG ∪ {u(1,a) = a, u(1,b) = b}	n(1,f(a,b)) = 1
p04d	LOG ∪ {n(1,a) = a, n(1,b) = b}	u(1,f(a,b)) = f(a,b)
p05a	LOG ∪ {u(1,a) = 1, u(1,i(a)) = 1}	1 = a
p05b	LOG ∪ {n(1,a) = 1, n(1,i(a)) = 1}	1 = a
p06a	LOG ∪ {u(1,b) = b }	u(1,f(i(a),f(b,a))) = f(i(a),f(b,a))
p06b	LOG ∪ {n(1,b) = 1 }	n(1,f(i(a),f(b,a))) = 1
p06c	LOG ∪ {u(1,b) = b }	n(1,f(i(a),f(b,a))) = 1
p06d	LOG ∪ {n(1,b) = q }	u(1,f(i(a),f(b,a))) = f(i(a),f(b,a))
p07	LOG ∪ {i(f(x,y)) = f(i(y),i(x))}	f(c,f(u(a,b),d)) = u(f(c,f(a,d)),f(c,f(b,d)))

Table A.2: Problems in the domain of lattice ordered groups (continued)

Name	Axioms	Goal
p08a	LOG ∪ { u(1,a) = a, u(1,b) = b, u(1,c) = c}	u(n(a,f(b,c)),f(n(a,b), n(a,c))) = f(n(a,b),n(a,c))
p08b	LOG ∪ { n(1,a) = 1, n(1,b) = 1, n(1,c) = 1}	n(n(a,f(b,c)),f(n(a,b), n(a,c))) = n(a,f(b,c))
p09a	LOG ∪ { u(1,a) = a, u(1,b) = b, u(1,c) = c, n(a,b) = 1}	n(a,f(b,c)) = n(a,c)
p09b	LOG ∪ {n(1,a) = 1, n(1,b) = 1, n(1,c) = 1, n(a,b) = 1}	n(a,f(b,c)) = n(a,c)
p10	LOG	i(u(a,b)) = n(i(a),i(b))
p11	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x)) }	f(a,f(i(n(a,b)),b)) = u(a,b)
p12	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x)), n(a,c) = n(b,c) , u(a,c) = u(b,c)}	a = b
p12x	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x)), n(a,c) = n(b,c) , u(a,c) = u(b,c), i(n(x,y)) = u(i(x),i(y)), i(u(x,y)) = n(i(x),i(y))}	a = b
p17a	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	u(1,u(a,1)) = u(a,1)
p17b	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	n(1,u(a,1)) = 1
p18	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	u(i(a),1) = i(n(a,1))
p19	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	a = f(u(a,1),n(a,1))
p20	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	n(u(a,1),i(n(a,1))) = 1
p20x	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	n(u(a,1),u(i(a),1)) = 1
p21	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	f(u(a,1),i(n(a,1))) = f(i(n(a,1)),u(a,1))
p21	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x)), i(n(x,y)) = u(i(x),i(y)), i(u(x,y)) = n(i(x),i(y))}	f(u(a,1),i(n(a,1))) = f(i(n(a,1)),u(a,1))
p22a	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	u(u(f(a,b),1),f(u(a,1), u(b,1))) = f(u(a,1),u(b,1))
p22b	LOG ∪ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	n(u(f(a,b),1),f(u(a,1), u(b,1))) = u(f(a,b),1)

Table A.3: Problems in the domain of lattice ordered groups (continued)

Name	Axioms	Goal
p23	LOG $\cup$ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	u(f(a,b),1) = f(a,i(n(a,i(b))))
p23x	LOG $\cup$ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	u(f(a,b),1) = f(a,u(i(a),b))
p33	LOG $\cup$ {n(u(a,i(a)),u(b,i(b))) = 1}	f(a,b) = f(b,a)
p38a	LOG $\cup$ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	u(b,u(a,b)) = u(a,b)
p38b	LOG $\cup$ {i(1) = 1, i(i(x)) = x, i(f(x,y)) = f(i(y),i(x))}	n(b,u(a,b)) = b
p39a	LOG $\cup$ {u(a,b) = a}	u(i(a),i(b)) = i(b)
p39b	LOG $\cup$ {n(a,b) = b}	n(i(a),i(b)) = i(a)
p39c	LOG $\cup$ {u(a,b) = a}	n(i(a),i(b)) = i(a)
p39d	LOG $\cup$ {n(a,b) = b}	u(i(a),i(b)) = i(b)
p40a	LOG $\cup$ {u(1,x) = x}	f(a,b) = f(b,a)
p8_9a	LOG $\cup$ { u(1,a) = a, u(1,b) = b, u(1,c) = c, n(a,b) = 1, u(n(a,f(b,c)),f(n(a,b),n(a,c))) = f(n(a,b),n(a,c)) }	n(a,f(b,c)) = n(a,c)
p8_9b	LOG $\cup$ {n(1,a) = 1, n(1,b) = 1, n(1,c) = 1, n(a,b) = 1, n(n(a,f(b,c)),f(n(a,b),n(a,c))) = n(a,f(b,c))	n(a,f(b,c)) = n(a,c)

Table A.4: Problems in the domain of lattice ordered groups (concluded)

Example	**add**	**occnest**	**fifo**	**max**	**gt**
ax_antisyma	0.230	0.190	0.370	0.230	0.240
ax_antisymb	0.210	0.230	0.360	0.240	0.200
ax_glb1a	27.790	0.820	–	770.940	–
ax_glb1b	0.730	0.290	5.030	1.810	–
ax_glb1c	28.110	0.820	–	695.700	–
ax_glb1d	0.800	1.020	432.800	1.970	–
ax_glb2a	0.220	0.200	1.730	0.750	–
ax_glb2b	0.250	0.210	0.320	0.260	0.230
ax_glb3a	0.220	0.210	0.250	0.240	–
ax_glb3b	0.230	0.210	1.160	0.220	–
ax_lub1a	2.490	0.270	5.410	1.770	–
ax_lub1b	42.900	2.160	–	799.020	–
ax_lub1c	42.750	2.200	–	688.930	–
ax_lub1d	2.470	1.050	234.290	2.080	–
ax_lub2a	0.290	0.220	0.600	0.540	–
ax_lub2b	0.200	0.220	2.430	0.790	–
ax_lub3a	0.260	0.230	0.930	0.500	–
ax_lub3b	0.210	0.210	0.250	0.250	–
ax_mono1a	46.670	0.200	4.800	761.900	–
ax_mono1b	47.920	0.220	4.700	750.190	–
ax_mono1c	47.560	0.300	–	754.750	–
ax_mono2a	44.860	0.220	4.450	751.100	–
ax_mono2b	46.380	0.220	4.410	744.800	–
ax_mono2c	45.120	0.240	–	739.240	–
ax_refla	0.210	0.190	0.220	0.200	0.170
ax_reflb	0.160	0.170	0.220	0.190	0.180
ax_transa	0.460	1.450	–	1.890	–
ax_transb	0.430	1.450	–	1.850	–

Table A.5: Conventional experts for lattice ordered groups

Example	**add**	**occnest**	**fifo**	**max**	**gt**
`distrnu`	–	–	–	–	–
`distrun`	–	–	–	–	–
`lat1a`	6.550	0.270	4.520	2.190	–
`lat1b`	6.570	0.210	4.450	2.270	–
`lat2a`	9.310	24.160	–	2.700	–
`lat2b`	9.160	21.460	–	2.850	–
`lat3a`	56.670	0.250	5.030	3.530	–
`lat3b`	56.960	0.260	4.930	3.520	–
`lat4`	893.500	–	–	576.810	–
`p01a`	–	0.480	–	–	–
`p01b`	–	0.460	–	–	–
`p02a`	–	–	–	711.700	–
`p02b`	–	–	–	711.560	–
`p03a`	–	4.400	–	–	–
`p03b`	–	2.690	–	–	–
`p03c`	–	2.760	–	–	–
`p03d`	–	4.370	–	–	–
`p04a`	34.220	2.000	–	192.800	–
`p04b`	9.700	1.930	–	149.500	–
`p04c`	9.790	1.900	–	153.040	–
`p04d`	33.780	2.020	–	188.230	–
`p05a`	7.930	177.350	–	2.300	–
`p05b`	7.960	177.920	–	2.300	–
`p06a`	–	0.630	–	–	–
`p06b`	165.810	0.320	–	–	–
`p06c`	167.190	0.310	–	–	–
`p06d`	–	0.570	–	–	–
`p07`	5.100	0.190	0.330	1.820	–
`p08a`	–	–	–	–	–
`p08b`	–	–	–	–	–
`p09a`	219.480	20.470	–	–	–
`p09b`	224.350	52.330	–	–	–
`p10`	–	–	–	–	–
`p11`	–	–	–	–	–
`p12`	–	–	–	–	–
`p12x`	–	–	–	–	–
`p17a`	0.280	0.200	0.560	0.520	–
`p17b`	0.250	0.210	2.420	0.800	–
`p18`	–	–	–	–	–
`p19`	–	–	–	–	–
`p20`	–	–	–	–	–

Table A.6: Conventional experts for lattice ordered groups (continued)

Example	add	occnest	fifo	max	gt
p20x	–	–	–	–	–
p21	–	19.520	–	663.380	–
p21x	7.890	0.300	521.570	1.920	–
p22a	5.100	0.970	1.000	1.800	–
p22b	5.040	11.070	531.700	34.430	–
p23	–	–	–	–	–
p23x	4.770	0.260	2.500	1.630	–
p33	–	–	–	–	–
p38a	0.250	0.190	0.860	0.520	–
p38b	0.230	0.180	0.230	0.240	–
p39a	45.210	5.460	–	871.040	–
p39b	45.400	4.070	–	873.060	–
p39c	45.340	4.040	–	872.760	–
p39d	45.250	5.460	–	871.370	–
p40a	5.240	2.030	–	1.840	–
p8_9a	–	–	–	–	–
p8_9b	–	–	–	–	–
Successes	59	67	33	58	5
Failures	27	19	53	28	81
Accumulated Time	29564.4	19569.1	54784.8	42374.8	81001.0

Note: A dash (–) indicates that DISCOUNT did not find a proof in 1000 seconds. For the accumulated time 1000 seconds are assumed in these cases.

Table A.7: Conventional experts for lattice ordered groups (concluded)

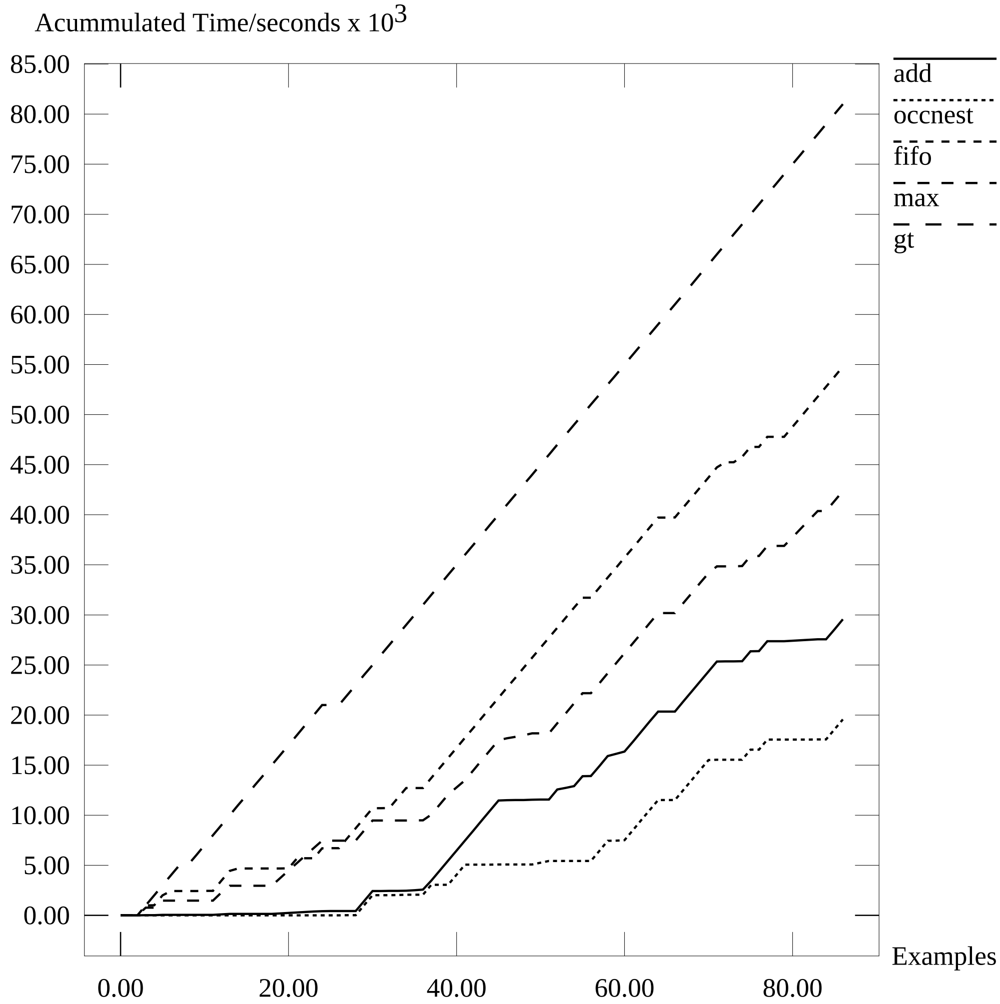


Figure A.1: Performance of conventional experts in the domain of lattice ordered groups

Appendix B

An Example Knowledge Base

In this chapter we will print some extracts of a large knowledge base for the domain of lattice ordered groups. It is the knowledge base used for the reproduction runs in the chapters 6 and 7, containing data from all the examples in appendix A we found any proof for. The complete knowledge base takes up slightly more than half a megabyte of space. Compared to the 50 megabytes a single complete PCL protocol can take up, the knowledge base is very compact indeed. 76 PCL listings have been inserted into the structure, with a total of 2830 steps, forming 44 goal domains and 61 specification domains. Despite this vast amount of data only 264 different representative generalizations are contained in the file `cdata` - a sign for both the power of the generalization approach and the importance of a limited number of results in the domain.

As already described in chapter 5, the knowledge base is realized by a file and directory structure in a UNIX file system.

B.1 The Directory `COMPLETE_KNOWLEDGE`

The presented knowledge base is named `COMPLETE_KNOWLEDGE` and forms a directory and file hierarchy with a root directory of the same name. The directory contains the files `goaldoms`, `specdoms`, `pdata` and `cdata`, and the subdirectories `EXAMPLES`, `GOALDOMS` and `SPECDOMS`.

B.1.1 Examples

Each knowledge base contains verbatim copies of all the pruned PCL listings inserted into it. We found this very valuable for a variety if reasons – ease of debugging and transparency are only two of them. The merging of two knowledge bases does not require any additional tools, but can simply be achieved by inserting the example files of one knowledge base into the other one.

All in all we found proofs for 76 of the 86 problems mentioned in appendix A. The corresponding files from the directory `EXAMPLES` follow:

```
ax_antisyma.xtr ax_lub3a.xtr    lat3a.xtr       p05b.xtr        p22a.xtr
ax_antisymb.xtr ax_lub3b.xtr    lat3b.xtr       p06a.xtr        p22b.xtr
```

```
ax_glb1a.xtr    ax_mono1a.xtr   lat4.xtr        p06b.xtr        p23x.xtr
ax_glb1b.xtr    ax_mono1b.xtr   p01a.xtr        p06c.xtr        p38a.xtr
ax_glb1c.xtr    ax_mono1c.xtr   p01b.xtr        p06d.xtr        p38b.xtr
ax_glb1d.xtr    ax_mono2a.xtr   p02a.xtr        p07.xtr         p39a.xtr
ax_glb2a.xtr    ax_mono2b.xtr   p02b.xtr        p08b.xtr        p39b.xtr
ax_glb2b.xtr    ax_mono2c.xtr   p03a.xtr        p09a.xtr        p39c.xtr
ax_glb3a.xtr    ax_refla.xtr    p03b.xtr        p09b.xtr        p39d.xtr
ax_glb3b.xtr    ax_reflb.xtr    p03c.xtr        p10.xtr         p40a.xtr
ax_lub1a.xtr    ax_transa.xtr   p03d.xtr        p11.xtr         p8_9a.xtr
ax_lub1b.xtr    ax_transb.xtr   p04a.xtr        p17a.xtr        p8_9b.xtr
ax_lub1c.xtr    lat1a.xtr       p04b.xtr        p17b.xtr
ax_lub1d.xtr    lat1b.xtr       p04c.xtr        p18.xtr
ax_lub2a.xtr    lat2a.xtr       p04d.xtr        p21.xtr
ax_lub2b.xtr    lat2b.xtr       p05a.xtr        p21x.xtr
```

The size varies from a mere 200 bytes for the trivial examples `ax_refla` and `ax_reflb` up to nearly 15 kilobytes for the example `p11` that was only proved using a team with learning experts. Due to the short size we can print `ax_refla.xtr` in its entirety:

```
 0 : tes-eqn : u(x,x) = x : initial
 15 : tes-goal : u(a(),a()) = a() : hypothesis
 16 : tes-rule : u(x,x) -> x : orient(0,u)
 17 : tes-final : a() = a() : tes-red(15,L,16,L)
```

B.1.2 Goal Domains

The file `goaldoms` contains the heads of the 44 goal domains formed by the insertion process. A short excerpt is printed below:

```
Domain: p40a

Type: Goal

Signature:
          f3 : 0
          f2 : 0
          f1 : 2

Examples: p40a

Goals:

f1(f2(),f3()) = f1(f3(),f2())
```

```
Domain: p39b

Type: Goal

Signature:
          f4 : 0
          f3 : 2
          f2 : 0
          f1 : 1

Examples: p39b, p39c

   .
   .
   .

Domain: ax_antisyma

Type: Goal

Signature:
          f2 : 0
          f1 : 0

Examples: p05a, ax_antisyma, ax_antisymb, p05b

Goals:

f1() = f2()
```

The directory `COMPLETE_KNOWLEDGE/GOALDOMS`

The complete domains are contained in the subdirectory `GOALDOMS`. One file represents each domain. They have been named automatically by the first example with a given goal:

```
ax_antisyma.gdm ax_lub3a.gdm    lat1a.gdm      p04a.gdm      p18.gdm
ax_glb1a.gdm    ax_lub3b.gdm    lat1b.gdm      p04b.gdm      p21.gdm
ax_glb1b.gdm    ax_mono1a.gdm   lat2a.gdm      p06a.gdm      p22a.gdm
ax_glb2a.gdm    ax_mono1b.gdm   lat3a.gdm      p06b.gdm      p22b.gdm
ax_glb2b.gdm    ax_mono2a.gdm   lat4.gdm       p07.gdm       p23x.gdm
ax_glb3a.gdm    ax_mono2b.gdm   p01a.gdm       p08b.gdm      p39a.gdm
ax_glb3b.gdm    ax_refla.gdm    p01b.gdm       p09a.gdm      p39b.gdm
ax_lub2a.gdm    ax_transa.gdm   p03a.gdm       p10.gdm       p40a.gdm
ax_lub2b.gdm    ax_transb.gdm   p03b.gdm       p11.gdm
```

For obvious reasons we only print one of the domains (selected for shortness). It is represented by the file `lat1b.gdm`.

```
Domain: lat1b

Type: Goal

Signature:
          f3 : 2
          f2 : 2
          f1 : 0

Examples: lat1b

Goals:

f1() = f2(f1(),f3(f1(),f1()))

Lemmas:

f3(f1(),x1) = f4(f3(f1(),x1),f3(f5(),x1)) : lat1b(1,6)
f3(f1(),x1) = f4(x1,f3(f1(),x1)) : lat1b(1,4)

Facts:

f4() = f2(f1(),f4()) : lat1b(1,9) , lat1b(1,10)
x1 = f4(x1,f2(x1,x2)) : lat1b(1,9) , lat1b(1,10)
f3(f1(),x1) = f4(f3(f1(),x1),x1) : lat1b(1,5)
x1 = f3(f4(),x1) : lat1b(1,6) , lat1b(1,7)
f1() = f1() : lat1b(0,0)
f4(x1,x2) = f4(x2,x1) : lat1b(1,5)
f3(f1(),x1) = f4(f3(f1(),x1),f3(f5(),x1)) : lat1b(1,6)
x1 = f2(x1,f4(x1,x2)) : lat1b(1,3) , lat1b(1,4)
x1 = f2(x1,f3(f1(),x1)) : lat1b(1,1) , lat1b(1,2)
f1() = f4(f1(),f5()) : lat1b(1,7) , lat1b(1,8)
f3(f1(),x1) = f4(x1,f3(f1(),x1)) : lat1b(1,3) , lat1b(1,4)
f3(f4(x1,x2),x3) = f4(f3(x1,x3),f3(x2,x3)) : lat1b(1,7) , lat1b(1,8)
```

B.1.3 Specification Domains

Similarly to goal domains, 61 specification domains were formed. Only very few examples had exactly the same specification. This is due to two facts. First, the original problem specifications (compare appendix A) in many cases used different additional axioms. These axioms usually were required for the proof and thus also distinguished the sets of axioms actually used. The second reason lies in the very nature of the domain. The specification is very rich, and only a few axioms are usually needed for the often small proofs. Thus, even problems with the same original specification happen to use different axioms.

The file `specdoms` contains the heads of the 61 domains:

```
Domain: lat4

Type: Specification

Signature:
          f7 : 1
          f6 : 1
          f5 : 1
          f4 : 2
          f3 : 0
          f2 : 2
          f1 : 2

Examples: lat4

Specification:

x1 = f1(x1,x1)
x1 = f2(f3(),x1)
x1 = f4(x1,f1(x1,x2))
f3() = f2(f5(x1),x1)
f6(x1) = f4(x1,f3())
f7(x1) = f1(x1,f3())
f1(x1,x2) = f1(x2,x1)
f4(x1,x2) = f4(x2,x1)
f1(x1,f1(x2,x3)) = f1(f1(x1,x2),x3)
f1(x1,f4(x2,x3)) = f4(f1(x1,x2),f1(x1,x3))
f2(x1,f1(x2,x3)) = f1(f2(x1,x2),f2(x1,x3))
f2(x1,f2(x2,x3)) = f2(f2(x1,x2),x3)
f2(x1,f4(x2,x3)) = f4(f2(x1,x2),f2(x1,x3))
f2(f1(x1,x2),x3) = f1(f2(x1,x3),f2(x2,x3))
f2(f4(x1,x2),x3) = f4(f2(x1,x3),f2(x2,x3))
f4(x1,f4(x2,x3)) = f4(f4(x1,x2),x3)

   .
   .
   .

Domain: ax_refla

Type: Specification

Signature:
```

```
          f1 : 2

Examples: ax_refla, ax_reflb

Specification:

x1 = f1(x1,x1)
```

The directory COMPLETE_KNOWLEDGE/SPECDOMS

As above, the complete data for specification domains can be found in a special subdirectory:

```
ax_antisyma.sdm ax_mono1b.sdm   p01a.sdm        p06a.sdm        p22b.sdm
ax_antisymb.sdm ax_mono1c.sdm   p01b.sdm        p06b.sdm        p23x.sdm
ax_glb1a.sdm    ax_mono2a.sdm   p02a.sdm        p06c.sdm        p39a.sdm
ax_glb1b.sdm    ax_mono2b.sdm   p02b.sdm        p06d.sdm        p39b.sdm
ax_glb1c.sdm    ax_mono2c.sdm   p03a.sdm        p07.sdm         p39c.sdm
ax_glb1d.sdm    ax_refla.sdm    p03b.sdm        p08b.sdm        p39d.sdm
ax_glb2a.sdm    ax_transa.sdm   p03c.sdm        p09a.sdm        p40a.sdm
ax_glb2b.sdm    ax_transb.sdm   p03d.sdm        p09b.sdm        p8_9a.sdm
ax_glb3a.sdm    lat1a.sdm       p04a.sdm        p10.sdm         p8_9b.sdm
ax_lub1a.sdm    lat1b.sdm       p04b.sdm        p18.sdm
ax_lub1c.sdm    lat2a.sdm       p04c.sdm        p21.sdm
ax_lub2b.sdm    lat2b.sdm       p05a.sdm        p21x.sdm
ax_mono1a.sdm   lat4.sdm        p05b.sdm        p22a.sdm
```

Again we print only one of the involved files. We use the domain created from the proof of the problem lat1b:

```
Domain: lat1b

Type: Specification

Signature:
          f5 : 0
          f4 : 2
          f3 : 2
          f2 : 0
          f1 : 2

Examples: lat1b, lat3b

Specification:
```

```
x1 = f1(f2(),x1)
x1 = f3(x1,f4(x1,x2))
x1 = f4(x1,f3(x1,x2))
f2() = f4(f5(),f2())
f3(x1,x2) = f3(x2,x1)
f1(f3(x1,x2),x3) = f3(f1(x1,x3),f1(x2,x3))

Lemmas:

f1(f5(),x1) = f4(x1,f1(f5(),x1)) : lat1b(1,4)
f1(f5(),x1) = f3(x1,f1(f5(),x1)) : lat3b(1,4)
f1(f5(),x1) = f4(f1(f5(),x1),f1(f2(),x1)) : lat1b(1,6)
f1(f5(),x1) = f3(f1(f5(),x1),f1(f2(),x1)) : lat3b(1,6)

Facts:

f5() = f5() : lat1b(0,0)
f2() = f3(f5(),f2()) : lat1b(1,9) , lat1b(1,10)
f6() = f6() : lat3b(0,0)
x1 = f3(x1,f1(f5(),x1)) : lat1b(1,1) , lat1b(1,2)
f2() = f4(f5(),f2()) : lat3b(1,9) , lat3b(1,10)
f5() = f4(f5(),f2()) : lat1b(1,7) , lat1b(1,8)
f4(x1,x2) = f4(x2,x1) : lat1b(1,5)
x1 = f4(x1,f3(x1,x2)) : lat3b(1,4) , lat3b(1,3) , lat1b(1,9) , lat1b(1,10)
x1 = f4(x1,f1(f5(),x1)) : lat3b(1,1) , lat3b(1,2)
f5() = f3(f5(),f2()) : lat3b(1,7) , lat3b(1,8)
f3(x1,x2) = f3(x2,x1) : lat3b(1,5)
f1(f5(),x1) = f4(f1(f5(),x1),x1) : lat1b(1,5)
x1 = f1(f2(),x1) : lat3b(1,7) , lat3b(1,6) , lat1b(1,6) , lat1b(1,7)
f1(f5(),x1) = f3(f1(f5(),x1),x1) : lat3b(1,5)
f1(f3(x1,x2),x3) = f3(f1(x1,x3),f1(x2,x3)) : lat3b(1,7) , lat3b(1,8)
x1 = f3(x1,f4(x1,x2)) : lat3b(1,10) , lat3b(1,9) , lat1b(1,3) , lat1b(1,4)
f1(f5(),x1) = f3(x1,f1(f5(),x1)) : lat3b(1,3) , lat3b(1,4)
f1(f5(),x1) = f4(x1,f1(f5(),x1)) : lat1b(1,3) , lat1b(1,4)
f1(f5(),x1) = f4(f1(f5(),x1),f1(f2(),x1)) : lat1b(1,6)
f1(f4(x1,x2),x3) = f4(f1(x1,x3),f1(x2,x3)) : lat1b(1,7) , lat1b(1,8)
f1(f5(),x1) = f3(f1(f5(),x1),f1(f2(),x1)) : lat3b(1,6)
```

B.1.4 Global Knowledge

Finally, the global knowledge used, for example, by the TET-based strategies, is found in the files `cdata` and `pdata`. Both follow the syntax description given in section 5.6. We only print a short excerpt from `cdata`:

```
Brain: complete_database
```

```
Type: Complete

Examples: p8_9a, p39d, p39b, p38b, p23x, p22a, p21, p17b, p11, p09b,
p08b, p06d, p06b, p05b, p04d, p04b, p03d, p03b, p02b, p01b, lat4,
lat3a, lat2a, lat1a, ax_transa, ax_refla, ax_mono2b, ax_mono1c,
ax_mono1a, ax_lub3a, ax_lub2a, ax_lub1c, ax_lub1a, ax_glb3a, ax_glb2a,
ax_glb1c, ax_glb1a, ax_antisyma, ax_antisymb, ax_glb1b, ax_glb1d,
ax_glb2b, ax_glb3b, ax_lub1b, ax_lub1d, ax_lub2b, ax_lub3b, ax_mono1b,
ax_mono2a, ax_mono2c, ax_reflb, ax_transb, lat1b, lat2b, lat3b, p01a,
p02a, p03a, p03c, p04a, p04c, p05a, p06a, p06c, p07, p09a, p10, p17a,
p18, p21x, p22b, p38a, p39a, p39c, p40a, p8_9b

Facts:

f1(x1,x2) = f2(f1(x1,x2),f1(x1,f1(f3(),x2))) : p06c(1,5) , p06c(1,4) ,
                                               p06b(1,4) , p06b(1,5)
f1(x1,x2) = f1(x1,f1(f2(f3(),x1),x2)) : p08b(2,6) , p08b(1,7)

   .
   .
   .

f1(f2(x1,x2),x2) = f2(f1(x1,f3()),x2) : lat4(1,13) , lat4(1,17)
f1(x1,f2(x1,x2)) = f1(x1,f2(f2(x1,x2),x3)) : p08b(1,24)
f1(x1,f2(x1,x2)) = f1(x1,f2(x1,f1(f3(),x2))) : p8_9b(1,10) , p8_9b(1,11) ,
                                               p8_9b(2,14) , p8_9b(1,15)
```

Appendix C

Implemented Programs

In addition to extensive changes and additions to the DISCOUNT system, two new programs for the management of the knowledge base had to be implemented. For use of the complete DISCOUNT system with all features a couple of additional programs have to incorporated, also. All programs mentioned have been implemented in ANSI C using the GNU C compiler. This choice ensures efficient code and a high portability.

C.1 Creating Knowledge Bases: `kb_create`

The program `kb_create` creates the file and directory structure representing an empty knowledge base. This structure consists of a main directory, two empty directories `SPECDOMS` and `GOALDOMS`, two empty files `specdoms` and `goaldoms` and files `cdata` and `pdata` containing nothing but the empty skeletons for the global knowledge store.

The program accepts one argument, the name of the knowledge base to be created. Only two options are supported, mainly to increase user-friendliness.

Option	**Effect**
`-h`	Prints the online documentation (shown below) and terminates the program.
`-v`	Prints the help page and a verbose description of the creation process.

If called with the option `-h`, the program prints the following explanations:

```
kb_create 1.0 (08.09.1994)

Usage: kb_create [-v] [-h] <name>

Written by Stephan Schulz

kb_create will create the directory designated by <name> and
initialize it for use as a knowledge base for DISCOUNT's learning
experts. <name> will be either an absolute or a relative path
name pointing to the directory to be created.
```

```
OPTIONS:
 -v     Print this information and comments on the operation's
        progress.
 -h     Prints this information and terminates.
```

C.2 Maintaining Knowledge Bases: kb_insert

The program kb_insert performs a much more complex job. It requires a knowledge base to be denoted by the -k option and reads a PCL file either from the standard input <stdio> or files given on the command line (multiple files are treated as a single PCL protocol). The program inserts the information contained in the PCL file into the knowledge base.

In a first phase it performs a pruning and lemma detection algorithm on the PCL protocol. Compare [DS94a] for details. Information from the pruned listing is then inserted into the knowledge base. The program uses the algorithms detailed in chapter 5 to insert all the equations into cdata, the lemmas into pdata and both equations and lemmas into the goal and specification domains. If a suitable domain does not yet exist, the program creates a new domain from the example. Finally, the pruned protocol is copied into the EXAMPLES directory.

The program accepts a variety of options to influence this process:

Option	Effect
-h	Prints the online documentation (shown below) and terminates the program.
-v	Prints a verbose description of the creation process, detailing the different operations.
-k <name>	Selects the name of the knowledge base the new knowledge shall be inserted into. This option is required, no default is assumed. Calling the program without the -k option generates an error message.
-e <name>	Sets the name, under which the current example is to be inserted into the knowledge base. This defaults to the base file name (without any suffix) of the PCL protocol to be inserted. If the protocol is read from <stdio>, this option is required.
-g <name>	Sets the name of the goal domain that will potentially be created. It defaults to the example name.
-s <name>	Sets the name of the specification domain that will potentially be created. It also defaults to the example name.

The online help of the program is given below:

```
kb_insert 1.0 1.0 (08.09.1994)

Usage: kb_insert[-v] [-h] -k <kb> ... [infile1 ... infileN]

Written by Stephan Schulz

kb_insert will try to insert the data from a PCL listing into the
```

```
knowledge-base denoted with the -k option.

OPTIONS:
 -v             Print a verbose protocol of the process.
 -h             Prints this information and terminates.
 -e <example>   Use <example> as the name for the example in the
                knowledge base. It defaults to the base name of the
                (first) input file. If neither is given the
                program will signal an error and terminate.
 -s <dom>       The name to use for a possible new specification
                domain to accomodate the example. It defaults to the
                example name.
 -g <dom>       The name to use for a possible new goal domain to
                accomodate the example. It defaults to the example
                name.
```

C.3 Other Useful Programs

`kb_create` and `kb_insert` are able to build and maintain a knowledge base without the use of other programs. However, for the handling of large proofs and for easy integration with the existing DISCOUNT system a couple of additional programs for the pruning process are useful.

For most cases the program `mextract` is more adequate to handle the pruning process. `mextract` is built around more efficient data structures for the internal representation of large PCL listings and, as a consequence, is able to handle protocols of difficult proofs much faster and easier than `kb_insert`. For even more extreme cases the combination of `rextract` and either `revert` or the GNU utility `tac` is useful. For a detailed description of these programs see [Sch93] and [DS94a].

Appendix D

Visible Changes to DISCOUNT

Several changes and additions to the DISCOUNT systems were necessary to implement the learning strategies. However, the outward behaviour of DISCOUNT has not changed very much. All changes are pure additions of new features. The behaviour for previously correct configurations did not change.

D.1 New Options for the Sequential System

The new evaluation strategies described in chapter 6 and—7 can be selected on the command line for sequential runs of DISCOUNT. This has been achieved by allowing new keywords for the argument of the `-x` or `-expert` option. The following arguments have been added:

Argument to -x	Selected strategy
`c_global_learn`	The reuse strategy `global_learn` as described in section 6.1.3, using the complete knowledge from the `cdata` file.
`p_global_learn`	The same strategy as above, but using only the lemmas stored in `pdata`.
`c_spec_bound`	The specification oriented reuse strategy from section 6.1.2, using the complete knowledge.
`p_spec_bound`	The same strategy, limited to the lemmas stored in the specification domain.
`c_goal_bound`	The goal oriented strategy `goal_bound_learn` from section 6.1.1, using all of the knowledge from an appropriate domain.
`p_goal_bound`	As the previous one, but limited to the lemmas.
`c_1tet_learn`	The first TET based strategy from chapter 7.
`c_2tet_learn`	The second of the TET based strategies.
`fifo`	The *fifo* strategy implemented for comparisons.

The new strategies require a number of parameters. For each of the parameters a command line option has been implemented. As usual with DISCOUNT options, a handy shortcut has also been provided. Each of this option takes a non-negative integer as a second argument.

Short	**Option**	**Parameter**	**Default**	**Comment**
`-T`	`-total_weight`	*total_w*	1	Compare page 58
`-A`	`-average_weight`	*average_w*	0	Compare page 58
`-R`	`-proofs_weight`	*proofs_w*	0	Compare page 58
`-G`	`-goal_dist_weight`	*goal_w*	1	Compare page 58
`-I`	`-learn_influence`	*learn_influence*	3	Compare page 58 and section 7.2
`-U`	`-unknown_weight`	*unk_penalty*	3	Compare section 7.2

The `-D -goal_dist_strat` option with the shortcut `-D` discriminates among the 4 goal distance evaluation strategies. It accepts arguments ranging from 0 to 3, with 0 selecting *aggressive*, 1 choosing *opportunistic*, 2 corresponding to *adaptive*, and 3 indicating to *conservative*. Default is *aggressive*.

Finally, the `-K` or `-knowledge_base` option selects the knowledge base to be used. If no `-K option` is given, `KNOWLEDGE` in the current working directory is assumed.

D.2 New Experts for the Team

As in former versions of DISCOUNT, the command line options influencing the critical pair evaluation strategy are ignored. The system reads all the necessary information from a central *configuration* file. This file contains informations about the experts and referees to use, a as well as information for the supervisors planning component. The configuration has been extended by one entry, denoting the knowledge base to use for the learning experts.

Experts for DISCOUNT are described by expert files. All expert files are collected in a single directory, one file per expert. Expert files contain, among secondary information, the description of the critical pair evaluation function, a critical goal evaluation function and a suggestion for the referee. The syntax has been extended to allow the specification of the new critical pair evaluation strategies. One expert for each of the strategies has been implemented.

The following new keywords are legal for the `CP_FUNKTION`[1] entry of an expert:

Keyword	**Expert**
`GLOBAL_LEARN`	`global_learn`
`GOAL_BOUND_LEARN`	`goal_bound_learn`
`SPEC_BOUND_LEARN`	`spec_bound_learn`
`1TET_LEARN`	`1tet_learn`
`2TET_LEARN`	`2tet_learn`
`FIFO`	*fifo*

Each of these keywords takes a number of qualifiers determining the value of the associated parameters. The qualifiers for the new strategies are detailed here:

`All_DATA`: A boolean value switching between using the complete knowledge or using only the lemmas.

`RED_INST`: A boolean value, determining, whether rules and equations generated by interre-

[1] For historical reasons some of DISCOUNT's configuration files still use German entries. This will change shortly.

duction are to be prefered to new critical pairs (a standard qualifier for all of DISCOUNT's experts).

DIST_STRAT: Determines the evaluation of the goal distance.

TOT_WEIGHT: Sets the parameter $total\_w$.

AVE_WEIGHT: Sets the parameter $average\_w$.

PROOFS_WEIGHT: Sets the parameter $proofs\_w$.

UNKNOWN_PENALTY: Sets the parameter $unk\_penalty$.

LEARN_INFLUENCE: Sets the parameter $learn\_influence$.

GOAL_DIST_WEIGHT: Sets the parameter $goal\_w$.

CLEVER_BACKUP: A boolean value, determining, whether `global_learn` or *add* is used as a backup strategy for `goal_bound_learn` and`spec_bound_learn`.

ALL_DOMAINS: A boolean value, determining wheter only the first matching specification is used for the specification bound experts or whether all matching domains are used.

We will print only one of the new experts here as an example.

D.2.1 The Expert C_GLOBAL_LEARN

This expert implements the reuse strategy `global_learn` detailed in section 6.1.3.

```
EXPERTENNAME                 C_GLOBAL_LEARN

EXP_OR_SPEC                  E

CP_FUNKTION                  GLOBAL_LEARN
                             ALL_DATA         : 0,1    ; 1
                             RED_INST         : 0,1    ; 0
                             DIST_STRAT       : 0,3    ; 2
                             TOT_WEIGHT       : -1000,1000 ; 1
                             AVE_WEIGHT       : -1000,1000 ; 2
                             PROOFS_WEIGHT    : -1000,1000 ; 1
                             GOAL_DIST_WEIGHT : -1000,1000 ; 1


CG_FUNKTION                  DIFF_GOAL
                             F_GEWICHT    : 1,5 ; 2
                             V_GEWICHT    : 1,5 ; 1
                             MIN_STRUCT   : 0,5 ; 4

FAIRNESS                     Y
```

```
GUTE_GUTACHTER             (STATISTIC, ,1)

GUTE_DOMAENEN

ROBUSTHEIT                 0.8

WISSENSANTEIL              1.0

BEWEISPHASE                ANFANG  1.0
                           MITTE   1.0
                           ENDE    0.5

ZYKLUSDAUER                ANFANG  (1,10)
                           MITTE   (1,10)
                           ENDE    (1,10)

AEHNLICHE_EXP

NACHFOLGER                 (GOALMATCH, RED_INST     : 0,1  ; 0
                                       NO_MATCH     : 0,1000 ; 10
                                       DOUBLE_MATCH : 0,100 ; 1
                                       SINGLE_MATCH : 0,100 ; 50, 0.5 )


UNVERTRAEGL

MOEGL_TEAMEXP

PROZEDUR

BESCHREIBUNG
```

D.3 Other changes

Most of the other changes to DISCOUNT are only visible to the programmer, not to the user of the system. They are summed up in short to aid future maintenance work.

- The last version of DISCOUNT required 4 executables to cover all combinations of protocol generation and reproduction. Now the reproduction version and the from scratch prover share a common executable, brining the total number down to two. The program checks, whether it is called under a name staring with `r` or `R` and, in these cases, switches to reproduction mode automatically.
- The `Makefile` for the system has been modularized. Shared parts for the two different executables now reside in a common file. This eases maintenance and eliminates a source for a particularly unpleasant class of errors.

- The `Makefile` has been extended by powerful targets for installation and de-installation. A complete distribution of the system can be build by a single `make` command.
- In addition to the source tree, the object files for the two different version are now kept in separate object directories, facilitating a fast recompilation after small changes.

Bibliography

[AA90] **Anantharaman, S; Andrianarievelo,N:** *Heuristical Criteria in refutational theorem proving*, Proc. 1st DISCO 1990, Capri, pp. 184-193, Springer Verlag, LNCS 429, 1990

[AD93] **Avenhaus, J.; Denzinger, J.:** *Distributing equational theorem proving*, Seki-Report SR-93-06, also in Proc. of the RTA-93, Montreal, pp. 62-76, Springer Verlag, LNCS 690, 1993

[Av91] **Avenhaus, J.:** *Reduktionssysteme I (german language)*, Skript zur Vorlesung im WS90/91, Department of Computer Science, University of Kaiserslautern, 1990

[Av90] **Avenhaus, J.:** *Reduktionssysteme II (german language)*, Skript zur Vorlesung im SS89, Department of Computer Science, University of Kaiserslautern, 1989

[BDP89] **Bachmair, L.; Derschowitz, N.; Plaisted, D.A.:** *Completion without Failure*, Coll. on the Resolution of Equations in Algebraic Structures, Austin 1987, Academic Press, 1989

[Bh90] **Bhatagar, N.:** *Adaptive Search by Explanation-Based Learning of Heuristic Censors*, Proc. 8th NCAI, Boston, pp. 895-901, MIT-Press, 1990

[BH93] **Bonacina, M. P., Hsiang, J.:** *Distributed Deduction by Clause-Diffusion: the Aquarius Prover*, Proc. 3rd DISCO 1993, pp. 272-287, Gmunden, Springer Verlag, LNCS 722, 1993

[CE+91] **Carbonell, J.; Etzioni, O.; Gil, Y.; Joseph, R.; Knoblock, C.; Minton. S.; Veloso, M.:** *PRODGY: An Integrated Architecture for Planning and Learning*, ACM SIGART Bulletin volume 2(4), pp. 51-55, 1991

[CN89] **Clark, P.; Niblett, T.:** *The CN2 Induction Algorithm*, Machine Learning, Volume 3, Kluwer, Boston, 1989

[Da+94] **Dahn, B.I.; Gehne, J.; Honigmann, T.; Walther, L.; Wolf, A.:** *Integrating Logical Functions with ILF*, Internal report, Institut für Reine Mathematik, Humbold-University, Berlin, 1994

[DCE81] *Dictionary of Contemporary English*, Langenscheid/Longman, Gütersloh, 1981

[De93] **Denzinger, J.:** *Teamwork: Eine Methode zum verteilten, wissensbasierten Entwurf von Theorembeweisern (german language)*, Ph.D.-Thesis, University of Kaiserslautern, 1993

[DF94] **Denzinger, J.; Fuchs, M. :** *Goal Oriented Equational Theorem Proving Using Team Work*, Proc. KI94, Saarbrücken, pp. 343-354, Springer Verlag, LNAI 861, 1994, also as SEKI-Report SR-94-04, Kaiserslautern, 1994

[DK94] **Denzinger, J.; Kronenburg, M:** *Planning for Distributed Theorem Proving: The Team Work Approach*, Seki-Report SR-94-09, Kaiserslautern, 1994

[DP92] **Denzinger, J.; Pitz, W.:** *Das DISCOUNT-System Benutzerhandbuch (german language)*, Seki-Working Paper SWP-92-16, 1992

[DS94a] **Denzinger, J.; Schulz, S.:** *Analysis and Representation of Equational Proofs Generated by a Distributed Completion Based Proof System*, Seki-Report SR-94-05, Kaiserslautern, 1994

[DS94b] **Denzinger, J.; Schulz, S.:** *Recording, Analyzing and Presenting Distributed Deduction Processes*, Proc. 1st PASCO, Hagenberg/Linz, pp. 114-123, World Scientific Publishing, 1994

[Et90] **Etzioni, O.:** *Why PRODIGY/EBL works*, Proc. of the 8th NCAI, Boston, pp. 916-922, MIT Press, 1990

[Et93] **Etzioni, O.:** *A Structural Theory of Explanation-Based Learning*, Artificial Intelligence, Volume 60(1), pp. 93-139, Amsterdam, North-Holland, 1993

[FHN72] **Fikes, R.E.; Hart, P.E.; Nilsson, N.J.:** *Learning and Executing Generalized Robot Plans*, Artificial Intelligence, Volume 3, pp. 251-289, Amsterdam, North-Holland, 1972

[FN71] **Fikes, R.E.; Nilsson, N.J.:** *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence, Volume 2, pp. 189-208, Amsterdam, North-Holland, 1971

[Fu94] **Fuchs, Matthias:** *The Application of Goal-Oriented Heuristics for Proving Equational Theorems via the Unfailing Knuth-Bendix Completion Procedure. A Case Study: Lattice Ordered Groups*, Seki-Report SR-94-02, Kaiserslautern, 1994

[KB70] **Knuth, D.E.; Bendix, P.B.:** *Simple Word Problems in Universal Algebras*, Computational Problems in Abstract Algebra, ed.: J.Leech, pp. 263-297, Pergamon Press, 1970

[KME91] **Knoblock, C.A.; Minton, S.; Etzioni, O.:** *Integrating Abstraction and Explanation-Based Learning in PRODIGY*, Proc. of the 9th NCAI, Anaheim, pp. 541-546 AAAI Press, 1991

[Kn73] **Knuth, D.E.** *The Art of Computer Programming, Volume 3 / Sorting and Searching*, Addison-Wesley Publishing Company, Reading, MA, 1973

[Kn94] **Knoblock, C.A.:**, *Automatically Generating Abstractions for Planning*, Artificial Intelligence, Volume 68, pp. 243-302, Amsterdam, North-Holland, 1994

[Ko87] **Kolodner, J.L.:** *Extending Problem Solver Capabilities through Case Based Inference*, Proc. of the 4th International Workshop on Machine Learning, Los Altos, 1987

[KW94] **Kolbe, T.; Walther, C.:** *Reusing Proofs*, Proc. of the 11th. ECAI, Amsterdam, pp. 80-84, 1994

[LG90] **Laird, G.; Gamble, E.:** *Extending EBG to Term Rewriting Systems*, Proc. 8th NCAI, Boston, pp. 929-935, MIT-Press, 1990

[Li93] **Lind, J.:** *Sicheres Broadcasting für DISCOUNT (german language)*, Project-Report, University of Kaiserslautern, 1993

[LO82] **Lusk, E.L.; Overbeck, R.A.:** *A Short Problem Set for Testing Systems that Include Equality Reasoning*, Argonne National Laboratory, Illinois, 1982

[LW92] **Lusk, E.L.; Wos, L.:** *Benchmark Problems in Which Equality Plays the Major Role*, Proc. of CADE-11, pp. 781-785, Springer Verlag, LNAI 60, 1992

[MC+87] **Minton, S.; Carbonell, J.G.; Etzioni, O.; Knoblock, C.A.; Kuokka, D.R.:** *Acquiring Effective Search Control Rules: Explanation-Based Learning in the PRODIGY System*, Proc. of the 4th International Workshop on Machine Learning, Irvine, pp. 122-133, Morgan Kaufmann, 1987

[MC+89] **Minton, S.; Carbonell, J.G.; Knoblock, C.A.; Kuokka, D.R.; Etzioni, O.; Gil, Y.:** *Explanation-Based Learning: A Problem Solving Perspective*, Artificial Intelligence, Volume 40 (Special Volume: Machine Learning), Amsterdam, North-Holland, 1989

[Mi88] **Minton, S.:***Quantitative Results Concerning the Utility of Explanation-Based Learning*, Proc. of the 7th NCAI, Saint Paul, pp. 564-569, Morgan Kaufmann, 1988

[MKK86] **Mitchel, T.M.; Keller, R.M.; Kedar-Cabelli, S.T.:** *Explanation Based Generalization: A Unifying View*, Machine Learning, Volume 1, Kluwer, Boston, 1986

[ML78] **Michalski, R.S.; Larson, J.B.:** *Selection of the Most Representative Training Examples and Incremental Generation VL1 Hypotheses: The Underlying Methodology and the Description of Programs ESEL and AQ11*, Technical Report 867, Department of Computer Science, University of Illinois, 1978

[Ox91] *Concise Oxford Dictionary, 8th Ed.*, Oxford University Press, Oxford, 1991

[Pi92] **Pitz, W.:** *Realisierung eines Systems zum verteilten, wissensbasierten Gleichheitsbeweisen mit Hilfe der Teamwork Methode (german language)*, Diploma thesis, University of Kaiserslautern, 1992

[Qu86] **Quinlan, J.R.:** *Induction of Decision Trees*, Machine Learning, Volume 1, Kluwer, Boston, 1986

[Ri89] **Richter, M.M.:** *Prinzipien der Künstlichen Intelligenz: Wissensrepräsentation, Inferenz und Expertensysteme (german language)*, Teubner Verlags-GmbH, Stuttgart, 1989

[RL86] **Rosenbloom, P.S.; Laird, J.E.:** *Mapping Explanation Based Generalization onto SOAR*, Proc. NCAI86, Menlo Park, 1986

[RM86] **Rumelhart, D.E.; McClelland, J.L. (and the PDP Research Group):** *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, Cambridge, MA., 1986

[RW90] **Richter, M.M.; Wendel, O.:** *Lernende Systeme I: Symbolische Methoden (german language)*, Skript zur Vorlesung im WS90/91, Department of Computer Science, University of Kaiserslautern, 1990

[Sch93] **Schulz, S.:** *Analyse und Transformation von Gleichheitsbeweisen (german language)*, Project-Report, University of Kaiserslautern, 1993

[Si92] **Sims, C.C.:** *The Knuth-Bendix Procedure for Strings as a Substitute for Coset Enumeration*, Journal of Symbolic Computation, Volume 12, 1991

[Ta56] **Tarski, A:** *Logic, Semantics, Metamathematics*, Oxford University Press, 1956

[VC91] **Veloso, M.; Carbonell, J.:** *Learning by Analogical Replay in PRODIGY. First Results*, Proc. EWSL91, Porto, Springer Verlag, LNAI 482, 1991

[Wu92] **Wusteman, J.:** *Explanation-Based Learning: A Survey*, Artificial Intelligence Review, Volume 6, Dordrecht, 1992