Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts

Jörg Denzinger[1] and Stephan Schulz[2]

[1] Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, D-67663 Kaiserslautern, Germany
[2] Institut für Informatik, Technische Universität München, Arcisstrasse 20, D-80290 München, Germany

Abstract. We present two inference control heuristics for equational deduction that are based on the evaluation of previous successful proof attempts in domains of interest. The first evaluation function works by symbolic retrieval of generalized patterns from a knowledge base, the second function compiles the knowledge into abstract term evaluation trees. Both heuristics have been implemented into the distributed equational proof system DISCOUNT. We analyze the performance of the heuristics on several sets of examples (including the subset of all unit-equality problems from the TPTP collection) and demonstrate their usefulness.

1 Introduction

The last years have seen a steady increase in the power of automatic theorem provers. There are a couple of reasons for this trend. The most significant ones are hardware improvements, refined inference engines, and stronger guiding heuristics. However, despite these advances, theorem provers still cannot rival humans. They perform basic inferences much faster than mathematicians, but are unable to cope with the complexities of the search space for many interesting problems.

One possible solution to this problem is to enable automatic systems to acquire knowledge about good heuristics by learning from examples. This resembles human behaviour: Basic operations of the calculus are gained first, then proof strategies are learned from presented examples and own experiences. In this paper we address this approach for an equational theorem prover. The required concepts are introduced in section 2: Unfailing completion ([HR87, BDP89]), the basic control strategies for the search process, the TEAMWORK method for distributing search processes ([De95, DFF97]), and proof recording and extraction ([DS94a, DS94b, DS96a]).

The task of learning heuristics for theorem provers is highly non-trivial. The first problem is the selection of the choice points to be controlled by a learning heuristic. If this selection has been made, training examples of good and bad decisions have to be obtained. In particular, it is necessary to distinguish between good and bad decisions and to find an adequate representation for these decisions. We solve this problem by analyzing proof protocols and by representing knowledge about proofs in the form of generalized *annotated patterns* of facts contributing to proofs. These problems are discussed in section 3.

A second set of problems concerns the usage of the learned knowledge.These problems include the selection of relevant parts of the knowledge for a given proof problem, dealing with the inherently incomplete and approximate nature of the knowledge in theorem proving, and in particular the need to fill gaps in this knowledge. To achieve the desired high degree of automatization, none of these problems can be relegated to user interaction. We solve these problems by using control heuristics that extend conventional general purpose control strategies, by structuring the knowledge base with respect to domains of interest (characterized by axiomatization) and goals to be solved, and by employing the TEAMWORK distribution method to combine different control heuristics. Section 4 discusses these aspects.

In section 5 we provide experimental results of our learning heuristics on two sets of examples: a collection of problems from the domain of lattice ordered groups and the full unit-equality subset of the TPTP library [SSY95]. In section 6 we will sketch some future work that will enable us to improve on the flexibility and expressibility of our heuristics.

2 Equational Theorem Proving

2.1 Unfailing Completion

We introduce some concepts of equational reasoning, using standard notations. The reader is assumed to be familiar with rewriting techniques.

An equational theorem prover is a system trying to deal with the following problem: Given a set $\mathbf{E}$ of equations, is an equation $s = t$ a logical consequence of $\mathbf{E}$ (written as $s =_{\mathrm{E}} t$)?

A completion based prover tries to decide the $\mathbf{E}$-equality of two terms by generating a (ground) confluent and terminating system of rules and equations for $\mathbf{E}$. Then any valid equation can be proved by reducing both sides into a common normal form. By following certain fairness criteria one can guarantee that any valid equation will be proved by an intermediate system after only a finite number of inference steps.

Our DISCOUNT system ([DKS97]) is based on the inference rules for *unfailing completion* ([BDP89, HR87]). We use three sets of term pairs to represent the current state of a completion process: A set $\mathbf{E}$ of processed, but unorientable equations, a set $\mathbf{R}$ of rules (processed and oriented equations) and a set $\mathbf{CP}$ of unprocessed equations. Rules are generated by orienting equations according to a *ground reduction ordering* $>$. New equations are generated by building *critical pairs* between existing rules and equations.

The completion algorithm will start out with empty sets $\mathbf{R}$ and $\mathbf{E}$, and the initial axioms in $\mathbf{CP}$. It will examine each equation in $\mathbf{CP}$, reduce it to normal form with respect to $\mathbf{E}$ and $\mathbf{R}$, use it to build new critical pairs (to be added to $\mathbf{CP}$) and to eliminate redundancies from $\mathbf{R}$ and $\mathbf{E}$ by simplification. It will then be added to either $\mathbf{R}$ (if it can be oriented according to $>$) or $\mathbf{E}$.

To build a prover on top of this algorithm the goal is reduced to normal form with respect to each successive $\mathbf{E}$ and $\mathbf{R}$. If these normal forms are identical or

subsumed by an equation from **E**, the goal is proved. Both completeness and efficiency of the proof process depend on the order in which the equations from **CP** are considered, with both goals often conflicting.

DISCOUNT allows the distribution of the search process on several computers. The main features of the used distribution scheme, TEAMWORK (see [DKS97, De95]), are the competition and cooperation of different selection functions for unprocessed equations, each of which controls an incarnation of DISCOUNT running on a different computing node. TEAMWORK provides an ideal basis for learning theorem provers (see also [DFF97]).

2.2 Evaluation Functions

The selection of critical pairs is usually guided by an evaluation function for equations. Equations which receive a low value are preferred to equations with a higher one. In this section we will introduce a general purpose evaluation function that serves as a base for our learning selection heuristics, and will sketch some other ideas for learning heuristics from literature.

A very simple, but nevertheless quite successful idea for an evaluation function is to count the function symbols and variables in an equations (compare [Hu80]). This idea can be generalized to the following generic evaluation function.

Definition 1 : AddWeight. Let t be a term and *varweight* and *funweight* fixed parameters. Then the term weight Weight of t is defined by

$\mathsf{Weight}(t) = \mathit{varweight}$, if t is a variable,
$\mathsf{Weight}(t) = \mathit{funweight} + \mathsf{Weight}(t_1) + \ldots + \mathsf{Weight}(t_n)$, if $t \equiv f(t_1, \ldots, t_n)$.

The value of the evaluation function AddWeight for an equation s=t is defined by

$$\mathsf{AddWeight}(s = t) = \mathsf{Weight}(s) + \mathsf{Weight}(t).$$

We found it useful to set $\mathit{varweight} = 1$ and $\mathit{funweight} = 2$, thus preferring equations with more variables, because they can be used for simplification more often.

AddWeight and extensions of it provide a good basis for learning approaches to control the search of a theorem prover. In [Fu95a] previous proofs are used to adapt parameters for an extension of AddWeight by assigning different weights to different function symbols. In [Fu96] AddWeight provides a basic weight that is improved by criteria measuring similarity to facts from a source proof. Both approaches are very promising, but require a single suitable source example (with a matching signature) whose previous found proof is used to guide the search for another target example. Our approach uses multiple proofs for different problems.

2.3 The TEAMWORK Method

The TEAMWORK method is a knowledge-based distribution concept for a certain kind of search processes (see [De95]). Unfailing completion is a process of this kind. Our TEAMWORK-based system for unfailing completion employs four types of components that use different kinds of knowledge and work together following a very strict interaction scheme.

Experts are completion procedures using different selection heuristics for critical pairs, thus exploring different parts of the search space. *Specialists* classify sets of equations and can add known consequences of a recognized set. *Referees* judge whole sets of equations (e.g. the work done by an expert), as well as single equations. The *supervisor* periodically generates a new common starting point for the experts by integrating the search state of the best expert, the best equations of the other experts, and the results of the specialists. It can also select new experts and specialist for work in the team. This task is controlled by *reactive planning*, and allows the team to adapt itself to a given problem (see [DK96]).

The interaction of the components is organized as a cycle. In the working phase of this cycle, the experts and specialists work independently and in parallel. In the judgment phase, the first phase of a *team meeting*, the referees evaluate the results of the experts and the specialists. This again can be done in parallel. In the second phase of a team meeting, the planning phase, the supervisor receives the results of the specialists and the referee reports and generates the new starting state. It then uses information from the referee reports, results of the specialists, and planning knowledge to select the next team.

We found the TEAMWORK method to be an ideal foundation for theorem provers that want to use learned knowledge (see [DFF97]). The referees allow the judgment of the different parts of knowledge that are represented by the experts. Specialist can be used for checking whether a certain piece of knowledge can be applied in a given situation. The use of contradictory control knowledge is also possible, since multiple experts can be used and their work is evaluated. Teams can contain experts with strong general purpose heuristics. These experts provide the means to "glue together" the work of other, more specialized experts.

2.4 Recording and Extracting Proofs

One basic requirement for learning from successful proof attempts is the ability to analyze a proof in detail. Not every protocol method that is used by theorem provers provides enough insight for this case. We solved this problem as follows (also see [DS94b, DS96a]):

In default mode, our prover protocols only enough information to reproduce the exact proof search (even for the distributed system). When a detailed proof listing is required, the necessary information is gathered in a *reproduction run* based on the information from the previous run. During the reproduction run each inference that is made is recorded as a PCL-step[3] (and steps from the

[3] PCL (Proof Communication Language) is a format that provides enough information for detailed analysis.

different processors in a distributed run are ordered to eliminate all forward dependencies). The resulting file can serve as a base for learning approaches using both positive and negative examples. A complete protocol is, however, too large for manual analysis, and contains much information needed only in special cases. Thus, starting at the end of the listing, our proof extraction tool strips the original file of all steps not contributing to the proof. The resulting file is used in our learning approach (which requires positive examples only). It also can be transformed into a proof suitable for human understanding.

3 Knowledge Acquisition for Proof Heuristics

As we already stated in section 2.1, the order in which unprocessed equations (mainly critical pairs, but also initial axioms, and results from the interreduction process) are selected for processing is the single most important choice for the efficiency of a completion based proof system. The heuristics for the selection of critical pairs typically influence the time needed to find a proof by several orders of magnitude. Simple analysis and some experiments show us, that, given an optimal evaluation function for critical pair selection, even the most complex among our test problems can be solved in trivial amounts of time. In order to improve the overall efficiency of a proof system, it is therefore sufficient to improve the evaluation function used to select the next facts to process[4]. Concentrating on this single most important choice point in the completion algorithm has some advantages: The learning component has a very clean and well-defined interface to the inference engine. Changes to the internal structure of the proof system are not necessary. Examples for the learning algorithm can be easily generated. As only equations need to be evaluated, no context information is required. Equations useful in a proof are positive examples, while useless equations are negative examples. Positive examples can easily be weighted according to their importance in a proof.

The choice of a critical pair evaluation function as the target for learning also has some less advantageous consequences, for example,

the number of critical pairs evaluated during a typical proof run is very high. Thus, to avoid a large overhead for the learning strategy, a single evaluation has to be very cheap in terms of CPU time. This fact will influence our choice of the knowledge representation.

3.1 Representing Knowledge about Proofs

When deciding on an appropriate representation for the knowledge about proofs we have to keep the following points in mind.

1. Knowledge retrieval at run time should be cheap. As much work as possible should be done off-line, while building the knowledge base.

[4] Similar results can be found for resolution based provers and other systems.

2. Completion-based proof systems are usually limited by memory constraints. Most proof attempts fail not for lack of time, but for lack of memory. To avoid an additional burden on the available memory, the knowledge representation should be as compact as possible.
3. The knowledge should be stored in a generalized form and abstract from irrelevant details like the concrete signature used in a given proof attempt.

In our experience, the structure of an equation is much more important for the evaluation than the concrete function symbols and their role in the axiomatization[5]. We therefore decided to store knowledge about proofs in the form of *annotated patterns* of positive examples (equations that contributed to a proof) only. Storing negative examples was ruled out by the memory constraints. The number of unnecessary equations in a medium difficulty proof is typically about 3 orders of magnitude greater than the number of contributing equations (compare [DS94a] and [DS94b] for exact numbers). Patterns serve both as an easily computable abstraction of equations and allow us to implement efficient knowledge retrieval mechanisms.

Definition2 : Representative Term Patterns. Let t be a term over some signature. Let $T = \{f_1, f_2, \ldots\}$ and $X = \{x_1, x_2, \ldots\}$ be two disjoint ordered sets of symbols. The *representative pattern* of t with regard to T and X is constructed by substituting the function symbols and the variables in t with symbols from T and X, respectively, in their order of appearance. Thus, the representative pattern of the term $t \equiv f(g(x), g(y))$ with respect to T and X would be $t\prime \equiv f_1(f_2(x_1), f_2(x_2))$. We call the representative pattern of a term t $\mathtt{pat}(t)$.

The two orderings on T and X can be combined into a total ordering $>$ on $T \cup X$ by assuming $f_i > x_i$ for all $f_i \in T$ and all $x_i \in X$. This total ordering obviously induces a total lexicographical ordering $>_{pat}$ on term patterns as well[6]. $>_{pat}$ is extended to a quasi-ordering on all terms by assuming $s \geq_{pat} t$ iff $\mathtt{pat}(s) \geq_{pat} \mathtt{pat}(t)$ for arbitrary terms s and t. Using this ordering, we can extend the definition of representative patterns to term pairs (rules or equations).

Definition3 : Representative Patterns for Equations. Assume T and X as above. The *representative pattern* for an equation $s = t$ is generated by first orienting the equation according to $>_{pat}$ (on individual terms) and then substituting the original function symbols and the variables in $s = t$ with symbols from T and X as above. We again write $\mathtt{pat}(s = t)$ to denote the representative pattern of an equation.

[5] Later experiments confirmed this view and showed us that even stronger abstractions are possible. Compare section 4.2.

[6] For reasons of efficiency we use a slightly different ordering in the implementation, comparing first pre-calculated term weights and using the lexicographical ordering only to break ties.

Please note that the representative patterns for both terms and equations are unique, and that $\mathtt{pat}(s) = \mathtt{pat}(t)$ is, in most cases, different from $\mathtt{pat}(s = t)$. $>_{pat}$ is extended to patterns for equations by lexicographic composition. It remains total on these patterns. Thus the ordering can be used to efficiently implement data structures for the retrieval of patterns (see next section).

As we stated above, we aim at a compact knowledge representation. Thus, for each contributing equation we insert the corresponding pattern into the knowledge base, annotated with the proof it occurred in, the number of times the equation was used in this proof, and the number of inferences on the longest path from the first occurrence of the equation to the inference step deriving the goal. If the same pattern appears more than once, the resulting annotations are simply concatenated. Thus, our knowledge about proofs consists of a set of patterns for equations, annotated with information about the occurrence of the corresponding equations in the proofs known to the system. The number of different proofs an equation occurred in can be used as an indicator for its versatility (i.e. its ability to contribute to multiple proofs), while the total number of references to an equation is a measure for the overall importance of the equation in a domain. The inference distance to the goal provides a very abstract view on the role of an equation in the structure of the proofs. This view can be used to prefer different equations in different phases of a proof attempt.

3.2 The Structure of the Knowledge Base

The collected knowledge about many proofs is organized in a way to allow efficient retrieval and easy maintenance. For this purpose, we keep three different sets of annotated patterns and a list of the original proofs that were used to generate the knowledge base.

The first set contains patterns organized by theorem. All equations used in proofs of theorems with the same structure are inserted into a single subset indexed by the pattern representing the theorem. A second set of equations is indexed not by the theorem, but by the axiomatization of problems these equations were successfully used in. Finally, we keep a flat set of patterns which are not associated with particular characteristics of the proof they occurred in.

Each subset of patterns is represented by an AVL tree (a partially balanced binary search tree) built using the ordering $>_{pat}$. As $>_{pat}$ can be calculated very fast (using pre-calculated weights) and as AVL trees are very efficient for data retrieval, patterns can be found in the sets at very little cost in CPU time. Moreover, we are able to store pre-calculated trees in a way that allows rebuilding of the trees without balancing operations. Thus, even the time needed to read a part of the knowledge base from disk is minimized.

AVL trees store data with very little memory overhead. Thus, this organization of the knowledge allows us to conserve both CPU time and memory, fulfilling the two main constraints required for our system.

3.3 The Knowledge Acquisition Cycle

Most learning systems that aim at improving the performance of an agent can be described by a circular model. We will now present this *learning cycle* for the DISCOUNT system. Fig. 1 gives an overview of this model.

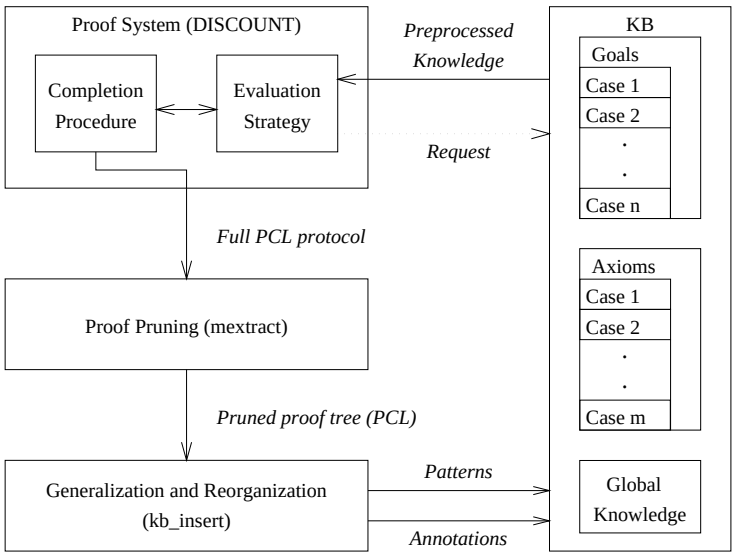

Fig. 1. Learning Model

In the first step, the prover is used to successfully prove a theorem, using either a learning strategy, a conventional one, or, in the case of a distributed TEAMWORK proof, a combination of multiple strategies, all of which can be either learning or conventional. The steps actually contributing to the proof are then extracted from the PCL protocol of the proof (see [DS94b, DS96a]). The equations used in these steps are generalized to their representative patterns and are inserted into the knowledge base as described in the previous sections. This updated knowledge base can now be used for proof reproduction purposes or to aid in the finding of new proofs.

4 Using the Knowledge Base

The previous section described the construction of the knowledge base. Now we present two very different strategies that make use of the accumulated knowledge. We will, in this paper, concentrate on strategies that use the whole knowledge base and make no use of the indexing by theorem or axioms. The indexing techniques only become important when the knowledge base grows significantly.

4.1 Simple Retrieval – Learning by Pattern Memorization

We call the first learning evaluation strategy `global_learn`. It searches the knowledge base for an entry equivalent to the representative pattern of the equation to be judged and uses the annotations to compute an evaluation.

Assume $s = t$ is the critical pair to be evaluated. The strategy assigns a weight to this critical pair as follows: If $\mathtt{pat}(s = t)$ is found in the knowledge base, then it returns the weight

$$W_{gl}(s = t) = \mathsf{AddWeight}(s = t) - (scale \times (to \times to_w + pr \times pr_w + av \times av_w + gd \times gd_w)) \quad (1)$$

where $\mathsf{AddWeight}(s = t)$ is the term weight of the equation (as described above), $scale$ is an arbitrary weighting factor (set to 20 in our implementation), and to, pr and av are the total number of times the fact has been referenced in any proof, the number of proofs it was used in, and the average number of applications it had in a proof. to_w, pr_w and av_w serve as adjustable weights for the previous three components.

By gd, with the weight gd_w, we want to represent the phase of the proof an equation was generated in. The longer the longest chain in a proof leading from an equation to the goal is, the more other equations are still needed to generate this particular proof. If we have to prove a given goal, and have among the critical pairs the two equations whose longest chain length is 1, then by selecting these two pairs the proof can be completed instantly. But typically, a pattern was used in several proofs and plays a different role in each proof. We base the evaluation of a pattern on the *average* distance to the goal agd over all proofs it appeared in.

Naturally, there are different ways to compute the value gd for a given pattern. In our experiments we used three different ways.

- We can try to force the selection of facts that appeared late in most proofs by using $gd = -agd$. Since large values of gd correspond to good evaluations (remember that we subtract from the value of $\mathsf{AddWeight}$), patterns that were often not far away from the goal will be preferred. This computation can be seen as the *aggressive* way, because it is intended to find short and fast proofs and totally neglects that the current goal may be different from all the goals that contributed to the value.
- We can try to estimate the distance of currently generated facts cgd and set $gd = -|cdg - agd|$, where cgd can be estimated by observing the average goal distance for the last couple of equations successfully found in the knowledge base. This computation concentrates on the idea of different phases in a proof and tries to select those equations that were used in the phases of the other proofs that are *analogical* to the phase the current proof attempt is in.
- Finally, we can try to prefer steps with a high goal distance to build a strong system of rules and equations before concentrating on the goal. This is achieved by setting $gd = -(max - agd)$, where max is the largest goal

distance in the knowledge base. We call this computation *conservative*, because it will usually build all equations with a large distance before moving closer to the goal.

If the pattern is not found, W_{gl} simply returns $\mathsf{AddWeight}(s = t) + pen$, where pen is a penalty large enough to ensure that critical pairs found in the knowledge base are nearly always preferred.

Thus, if all the weighting factors are positive numbers, the strategy will prefer known pairs to unknown ones, and it will furthermore show a preference for pairs that appeared more often in proofs covered by the knowledge base, and pairs preferred by the distance evaluation strategy.

We found this strategy to be excellent for the reproduction of proofs already covered by the knowledge base. It also leads to improved behaviour for many new, unknown proofs. However, this extrapolative power is most pronounced in cases where a lot of previous examples from a domain are known. It is quite limited if only a few examples have been inserted into the knowledge base. For experimental results see section 5.

4.2 Strong Abstraction – Learning by Term Space Mapping

The `global_learn` strategy uses learned knowledge only to evaluate term pairs that correspond to existing patterns. It uses a conventional back-up strategy in all other cases. We now present a strategy that uses the knowledge base to define measures of quality on *all* terms.

For the `1_tet_learn` strategy, the complete knowledge is compiled into a single *term evaluation tree* (*TET* for short). This TET is then used as a template or *map* to evaluate both sides of the equation to be judged. The results are combined into a single weight for the equation.

A TET represents a number of possible alternatives for the structure of terms. Each of these alternatives stands for all terms with a given number of principal subterms and is represented by a *term arity tree* or TAT. The alternatives for a given subterm are again represented by a TET. More exactly, TETs and TATs are defined by mutual recursion:

Definition 4 : TETs and TATs. A TET is a (possibly empty) set of TATs. A TAT is a tuple $(arity, count, info, (tet_1, \ldots, tet_{arity}))$, where the tet_i are non-empty TETs, and $arity$, $count$ and $info$ are elements from $\mathbb{N}$. We call $info$ an *annotation* to the tree node. $arity$ gives the arity of the TAT.

Only 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 5 : Inserting terms into TETs. Let $T = (tat_1, \ldots, tat_m)$ be a TET and let $t \equiv f(t_1, \ldots, t_n)$ be a ground term with n principal subterms. Assume $info \in IN$. We call a tuple $(t, info)$ a *term with additional data*. The function `ins` accepts a TET and a ground term with additional data and returns

a TET. If none of the tat_i has arity n, a new TAT is added and the subterms are inserted recursively:
$\texttt{ins}(T,(t,info)) := T \cup$
$\quad \{(n,1,info,(\texttt{ins}(\{\},(t_1,info)),\ldots,\texttt{ins}(\{\},(t_n,info))))\}$
Otherwise, there exists a $tat_i \equiv (n,count,info',(tet_1,\ldots,tet_n))$. In this case the annotation of the term is merged into the tat_i. Again, the subterms are inserted by recursion:
$\texttt{ins}(T,(t,info)) := \{tat_1,\ldots,tat_{i-1},tat_{i+1},\ldots,tat_m\} \cup$
$\quad \{(n,count+1,info+info'),(\texttt{ins}(tet_1,(t_1,info)),\ldots,\texttt{ins}(tet_n,(t_n,info)))\}$
`ins` is extended to non-ground terms by treating variables as constants.

Insertion of terms in TETs is associative, the resulting tree depends only on the terms (and data) inserted into it, not on the particular order of the terms. Thus, it makes sense to speak of the representative TET for a set of equations.

Definition 6 : Representative Term Evaluation Trees. Assume a finite set $S = \{(t_1,info_1),\ldots,(t_n,info_n)\}$ of terms with additional data. $\texttt{rep_TET}(S)$ is the unique TET generating by inserting the terms from S into an empty TET. For a set $D = \{(s_1 = t_1,info_1),\ldots,(s_n = t_n,info_n)\}$ of equations with data $\texttt{rep_TET}(D) = \texttt{rep_TET}(\{(s_1,info_1),(t_1,info_1),\ldots,(s_n,info_n),(t_n,info_n)\})$.

The `1_tet_learn` strategy builds the representative TET of the patterns found in the knowledge base, using the number of proofs the corresponding equations appeared in as additional data. This TET is used to evaluate equations using a number of measures.

Definition 7 : TET-defined term measures. Let $T = \{tat_1,\ldots,tat_m\}$ be a TET and let $t = f(t_1,\ldots,t_n)$ be a term whose top function symbol has arity n. T defines three measures on t: If there is no tat_i with arity n in T, then

- $\texttt{eval}_\texttt{W}(t,T) = pen_u \times AddWeight(t)$
- $\texttt{eval}_\texttt{C}(t,T) = 0$
- $\texttt{eval}_\texttt{I}(t,T) = 0$

Otherwise there is a $tat_i = (n,count,info,(tet_1,\ldots,tet_n)$ in the TET and

- $\texttt{eval}_\texttt{W}(t,T) = \sum_{i=1}^{n} \texttt{eval}_\texttt{W}(t_i,tet_i)$
- $\texttt{eval}_\texttt{C}(t,T) = count + \sum_{i=1}^{n} \texttt{eval}_\texttt{C}(t_i,tet_i)$
- $\texttt{eval}_\texttt{I}(t,T) = info + \sum_{i=1}^{n} \texttt{eval}_\texttt{W}(t_i,tet_i)$

The three measures calculate the weight of subterms not represented in the TET (with a penalty factor pen_u), the number of times an inserted term was mapped to TET nodes corresponding to the term and the sum over the information stored at these TET nodes, respectively.

The `1_tet_learn` strategy calculates the weight of a term from three components: First, the conventional term weight, then the weight of the term nodes not covered by the TET and finally the information collected. We empirically found the following equation to yield good results:

$$W_{tet} = AddWeight(s = t) + eval_W(s,T) + eval_W(t,T) - l_{infl} \qquad (2)$$

$$\times \lfloor \frac{\sqrt{p_w * (\mathtt{eval_I}(s,T) + \mathtt{eval_I}(t,T))}}{(s_{tet} * \mathtt{eval_C}(s,T) + 1 + \mathtt{eval_C}(t,T) + 1)} \rfloor$$

In this equation, l_{infl} is a weighting factor for the knowledge found in the TET, p_w is a weighting factor for the collected addition data (number of proofs) and s_{tet} is a factor for the number of terms inserted into the TET.

The strategy resulting from this weight function has excellent generalization capabilities. A few proofs will suffice to generate a strong heuristic. However, the strategy is less good at pure reproductive tasks and will break down if too much knowledge is used in the TET. The reasons for this are not yet clear and will be subject of further work. Experimental results will be presented in the next section.

5 Experiments

We have performed a variety of experiments to evaluate the learning strategies and some of their parameters. In the following, we will first concentrate on sequential test runs that allow us to compare our learning heuristics with DISCOUNT's non-learning ones and with the renown prover Otter (see [Mc94]). Afterwards we will provide experiments of DISCOUNT working in TEAMWORK mode and using our learning heuristics.

Problems for (equational) theorem provers are often generated in a rather *ad hoc* fashion. A large number of small, unrelated problem classes does (and did) exist, but there was a distinct lack of problems scalable to various difficulties, or of a domain with a variety of problems covering a wide range of difficulties.

However, we are now in a more fortunate position. The DISCOUNT system was, among other provers, used for the construction of a proof pad for mathematicians (the ILF project – see [Da+94]). ILF was used on the test case of lattice ordered groups. In the course of the evaluation of ILF, a large number of problems from the domain of lattice ordered groups have been generated. We have, for some time now, used these examples from a field of active interest to mathematicians in the evaluation of DISCOUNT. The examples used have also been accepted into the TPTP problem library ([SSY95]) and appear in the latest release.

Since the examples from the domain lattice ordered groups range from quite easy to hard they provide an optimal test set for learning theorem provers. By learning from the easy examples the prover should be able to solve some harder ones that then can be used to solve even harder ones and so on. But naturally, the performance of the learning experts in more general settings with multiple different domains is of interest. Therefore, we will first concentrate on the examples from the domain lattice ordered groups using different experimental settings, and then different parameter selections. After analyzing the obtained results we will then report the results of the interesting settings for the complete set of pure unit-equality problems from the TPTP problem library.

5.1 Methodology.

In contrast to conventional theorem provers that tackle each example in a set of example problems without being influenced by the experiences with other examples, the results generated by a learning prover depend on the ordering in which the prover is presented the examples, the strategy that decides which examples to learn from and the sources the prover can get proofs to learn from. Therefore, in our first set of experiments we used three different modes for our test runs in the domain lattice ordered groups.

In all modes, DISCOUNT tries to prove all of the 86 selected examples. The difference is the composition of the knowledge base. *Explorative* 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. A proof is added to the knowledge base regardless of whether the proof was found by the learning strategy or by any other means, including team runs (with and without learning team members) and proof attempts for more than 1000 seconds. Note that there are still 10 examples for which *no* proof was 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.

For comparison we will also present the results for two conventional strategies: The AddWeight strategy, which uses the term weight of an equation as an evaluation (compare section 2.2), and the Occnest strategy ([DF94]). The AddWeight strategy is a very strong general purpose strategy that also serves as the base for the learning heuristics. The Occnest strategy is a goal oriented heuristic hand-tuned for strong performance in the domain of lattice ordered groups.

The second set of experiments aims at the influence of some parameters of the heuristic `global_learn`. The most interesting parameter is the influence of the distance to the goal (for which we suggested three different ways to compute a value), which is the main focus of this set of experiments. Finally, the third set of experiments broadens the set of examples by using all the appropriate examples from TPTP.

5.2 Results and Discussion.

Results for the domain *lattice ordered groups* in different sequential modes. Table 1 shows the results of both the non-learning and the learning strategies in the three different modes for the domain lattice ordered groups. As additional

Strategy		Successes	Total time
AddWeight		59	29564
Occnest		67	19569
Otter		67	22901
global_learn[7]	(explorative)	58	28445
	(persistent)	67	19442
	(reproductive)	75	11266
1_tet_learn[8]	(explorative)	68	21833
	(persistent)	68	22607
	(reproductive)	57	31563

Table 1. Results on 86 problems in the lattice ordered groups domain

measure, we also included the results for Otter[9]. The results for the learning strategies in this table use static information about frequencies of patterns only, and ignore the goal distance. The first column gives the strategy and, for the learning strategies, the mode of learning. The second column gives the number of proofs found in 1000 seconds, and the final column gives the total CPU time for all examples, counting failures with 1000 seconds. All times are measured on a SPARCstation ELC.

Fig. 2 shows the performance of some selected strategies in more detail. In this figure we plot the total time spend on all examples versus the number of examples. The global_learn heuristic in explorative mode performs similar to the standard AddWeight. Except for one case, both strategies prove the same examples. In this one case, a very difficult example cannot profit from the preceding, easy examples, and the (quite small) overhead of the learning strategy increases the total proof time to over 1000 seconds. Despite this one case, the overall performance of global_learn is slightly better than the performance of AddWeight.

In the more realistic persistent mode, the general purpose global_learn strategy performs significantly better and even beats the specialized Occnest strategy by a small margin. Both strategies do not prove the same set of problems – each strategy proves 4 problems the other is unable to prove. In reproductive mode the strategy performs still better. It can reproduce 74 of the inserted proofs very fast (all 75 proofs were found in only 266 seconds), and even solves one example where no prior proof had been found by any strategy. Two of the proofs inserted cannot be reproduced within the time limit. It should be noted

[9] We used the autonomous mode of Otter, but the term ordering also given to DISCOUNT.The standard ordering chosen by Otter could not solve as many problems as the runs reported here.

[8] We used global_learn with $to_w = 20$, $av_w = 0$, $pr_w = 0$ and $gd_w = 0$ for these results.

[9] For 1_tet_learn the parameters were $l_{infl} = 10$, $p_w = 1$ and $pen_u = 5$.

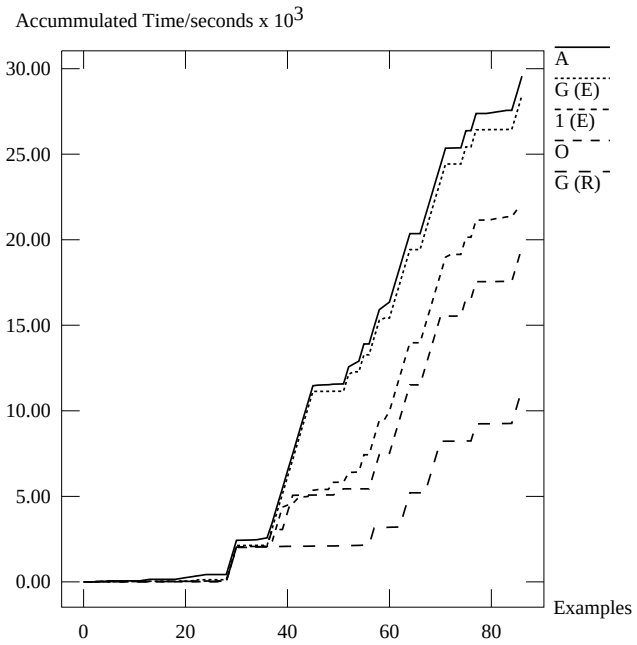

Fig. 2. Comparative behaviour of different strategies. Plotted are times for AddWeight (A), Occnest (O), `1_tet_learn` in extrapolative mode (I (E)) and `global_learn` in extrapolative (G (E)) and reproductive (G (R)) mode.

that only TET-based strategies have been able to find proofs for these problems.

We found that the strategy based on term pair retrieval works better with more data. If only a few examples are available, the strategy works for examples similar to the ones in the knowledge base because the known term pairs are preferred. If more examples are inserted into the knowledge base, the strategy learns to stronger differentiate among the term pairs that occurred in known proofs. In this case the strategy also gains extrapolative power. Fig. 2 shows this effect. At first, the graph for the learning strategy in explorative mode closely hugs the graph of AddWeight. Only with increasing knowledge the learning strategy outperforms the conventional one. This effect is even more pronounced for the persistent mode not included in the diagram.

The `1_tet_learn` strategy performs very well in both explorative and persistent mode. It is the strategy that finds the most proofs. Again, the examples found by this strategy are not a pure superset of the examples found by *Occnest*. The strategy solves 4 examples no other strategy is able to prove. However, the strategy performs much worse in reproductive mode. The current version of a

TET-based strategy cannot cope with too much knowledge. The performance graph of the strategy in extrapolative mode also demonstrates this. Most of its performance advantage is found near the middle of the graph, where enough knowledge for meaningful learning is available, not at the right side, where the strategies weakness for coping with too much knowledge becomes visible. We suspect one main reason for this weakness. Most terms inserted into the TET have a certain size limit. This leads to a very strong preference for terms below this size and thus makes the strategy overly conservative.

Influence of the goal distance in the domain *lattice ordered groups.* Table 2 illustrates the effect of taking the estimated distance of the equation to be evaluated to the goal into account. All results in this table were obtained using the `global_learn` strategy in reproductive mode, using the same knowledge base of 76 different problems (with one proof per problem) as before.

The first entry reproduces the result without using any goal influence (i.e. with $gd_w = 0$) from Table 1. The next 3 results used *only* information about the distance to the goal to differentiate among patterns. As one can see, in the domain of lattice ordered groups the *aggressive* strategy, i.e. the preferring of equations close to the goal, can significantly increase the efficiency of the prover. This strategy solves three examples not solved by any other learning strategy, although it still fails to prove the two examples provable only by TET-based strategies. The time needed to prove the examples that can be solved by the `global_learn` strategy without goal influence is for the aggressive strategy significantly smaller.

The results are even better when information about the frequencies of patterns in successful proofs is used in combination with the goal distance. However, the goal distance seems to be the most important factor. The strength of the aggressive strategy may have causal links with the strength of the *Occnest* strategy in this domain, which prefers equations similar to the goal.

Results of a Team. Table 3 reports the results of a team using our learning strategies (`global_learn`: reproductive mode, aggressive in using goal influence and frequency information; `1_tet_learn`: persistent mode) together with our standard strategies AddWeight and Occnest. At each time only two experts are active; which ones is decided by the supervisor (see [DK96]. While the comparison with the single runs with respect to successful proof attempts shows an improvement of 2 examples, the total time used by the team is significantly smaller than the time needed in the single runs.

If we subtract the 6000 seconds that are the result of the 6 unsuccessful runs (these six problems that could not be solved in any of our experiments are included in the appendix of this article), then we have 1625 seconds run time (measured as wall clock time) for the 80 problems (compared to 2395 seconds for the same problems needed by the best learning strategy).

A detailed analysis of the results showed that the team is much slower than the single strategies when proving easy examples (which was expected). Because

Version	Successes	Total Time
No goal influence		
	75	11266.1
Goal influence only		
aggressive	78	8621.96
analogical	75	11495.5
conservative	75	11915.9
Goal influence and frequency information		
aggressive	78	8395.1
analogical	75	11316.2
conservative	75	11717.6

Table 2. Influence of the goal distance for `global_learn` in the domain of lattice ordered groups

Successes	Total Time
80	7624.8

Table 3. Results for a team with learning experts

the team has to be started, connections between the two computers have to be established and the right experts have to become members of the team, we have no team run that does not need at least 4 seconds wall clock time. Single strategies need for easy examples less than a second. Since we use two computers we have a factor of at least 8 that the team is slower. If we analyze the team runs with hard examples that can be solved only by a few of the single strategies, then the cooperation between the learning strategies (and the standard strategies) pays of. We could observe factors between 4 and 10 that the team was faster than the best learning strategy.

An analysis of the generated proofs showed that for some examples only the two learning experts contributed to the proof while for other examples also the standard experts had to provide results. But there was no hard problem that was solved without contributions by the learning experts.

Since we used the best variant of the expert `global_learn` in reproductive mode the team did not have much room for improvements. However, the team was not only able to prove the two examples provable only by TET-based strategies, it also improved on those examples that caused problems for `global_learn`. It should be noted, that among the 76 problems we had proofs to include into the knowledge base were 7 problems which could only be solved by a team (without learning experts). Therefore we expect that the use of teams will be a necessity if one starts exploring a new domain of interest.

Results for TPTP. Table 4 shows the results the `global_learn` heuristic in reproductive mode achieved on the subset of pure unit-equality problems from the TPTP v. 1.2.1. There is a total of 381 problems in this category[10]. The knowledge base consisted of 792 proofs for 244 of these problems. About two thirds of these proofs have been generated using a lexicographical path ordering and various different proof strategies (including `global_learn`, AddWeight and Occnest). The remaining third of the proofs were obtained using the same Knuth-Bendix ordering used for the results presented.

There is a subclass of 80 unit equality problems which contain existentially quantified goals. For these problems no examples can be generated by our current implementation, although the learning strategies can still be used to guide the completion part of the proof search. The interleaved narrowing part necessary for these problems is guided by a conventional heuristic.

All these test runs were done on a SUN SPARCstation Ultra 2/200 with 512 megabytes of main memory. To avoid hiding the real differences for this large class of problems, only the times for successful proofs have been reported. Both pattern frequencies and goal distance were used for the evaluation of a pattern. The AddWeight strategy had a time limit of 500 seconds for each problem, Occnest and the learning strategies were limited to 250 seconds[11]. Despite this advantage, `global_learn` clearly outperforms the conventional strategies for both the aggressive and the analogical goal distance evaluation.

Strategy	**Successes**	**Time for Proofs**
AddWeight	251	4250.68
Occnest	243	3080.81
`global_learn` (*aggressive*)	272	2188.53
`global_learn` (*analogical*)	273	1882.69
`global_learn` (*conservative*)	215	2738.94

Table 4. Performance on the TPTP unit equality problems

It is interesting to note that in this case the analogical approach for the goal distance gives the best results. This may be due to the fact that much more proofs from multiple very different domains have been combined into a single knowledge base, so that a more careful building of the system of rules and equations becomes a necessity. The conservative evaluation function performs very bad in this case. It leads to a kind of depth-first search that is unsuitable

[10] Two of the TPTP unit equality problems contain three negative literals each. They have been split into three separate problems, with a different negative literal transformed into the equational goal for each new problem.

[11] Our experiments showed that Occnest finds a proof quite fast or not at all. Thus, longer time limits for Occnest are unlikely to change the picture.

to the quite large knowledge base for the whole TPTP.

To summarize, we found the learning strategies to be quite powerful and very valuable even for a sequential prover. Even in explorative mode the learning strategies were able to find 4 previously unknown proofs, reducing the number of unsolvable examples for the sequential prover in the domain lattice ordered groups from 18 to 14. Moreover, the experiments in persistent mode show that the learning strategies can even profit from proofs found by very different strategies. Finally, the performance of `global_learn` in reproductive mode shows that this strategy seems to scale very well and that further improvements can be expected for larger knowledge bases.

6 Conclusion and Future Work

We presented several selection heuristics for terms and term pairs that are based on term patterns and the evaluation of the usage of these patterns in several previous successful proof attempts for different problems. These selection strategies use different degrees of abstraction from the patterns and also utilize data about the proof phases a particular pattern was most often used in.

Our experiments show that annotated patterns carry sufficient information to represent good proof heuristics. Despite the abstractions and the combination of data from proofs for many different problems into a single strategy, the reproduction of training proofs is still possible. Even more, abstractions and combination allowed our learning prover to solve several new proof problems that were beyond the reach of the non-learning version of it.

An efficient access function for stored patterns allows a very cheap evaluation of terms even in large knowledge bases. Future work in this direction should focus on even larger, more complete collections of proofs. These tests will show whether filters for restricting access to a large knowledge base will become necessary.

The TET-based strategies, which abstract from most properties of terms, can still achieve good results. Moreover, they require only a few examples to induce a strong evaluation function. Further research will hopefully show us how to eliminate their weakness in cases where much knowledge is available.

Much of our future work in this field will concentrate on experimenting with different degrees of abstraction, in order to fill the large gap between pattern retrieval and term evaluation trees. Thus, we hope to improve the generalization ability of the pattern based strategies without loosing their ability to reproduce proofs. Complementary to this, we also hope to improve the reproductive ability of the TET-based strategies and to cure their weakness in dealing with too much knowledge. We also hope to find more canonical ways to compute evaluations from the retrieved knowledge, and thus to eliminate many of the parameters now used for the fine-tuning of the strategies. Currently, the parameter space of DISCOUNT is so large that there is little hope of exploring all possible or even interesting settings.

But our experiments also showed that there probably will not be one single (learning) selection strategy that is always successful. Instead, the different

strategies learn different aspects of the proofs that are presented to them. So, the selection of the right strategy to a given example remains a problem. Even more, complex examples will require the combination of different learned aspects in order to solve them. We hope to tackle these problems by including other learning heuristics into DISCOUNT, for example those of [DFF97], and use all of these learning experts together in teams.

References

[AD93] **Avenhaus, J.; Denzinger, J.:** *Distributing Equational Theorem Proving*, Proc. 5th RTA, Montreal, LNCS 690, pp. 62-76, 1993; also available as SEKI-Report SR-93-06, University of Kaiserslautern, 1993.

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

[De95] **Denzinger, J.:** *Knowledge-Based Distributed Search Using Teamwork*, Proc. ICMAS-95, San Francisco, AAAI Press, pp. 81-88, 1995.

[DF94] **Denzinger, J.; Fuchs, M.:** *Goal Oriented Equational Theorem Proving Using Teamwork*, Proc. 18th KI-94, Saarbrücken, LNAI 861, pp. 343-354, 1994; also available as SEKI-Report SR-94-04, University of Kaiserslautern, 1994.

[DFF97] **Denzinger, J.; Fuchs, Matt.; Fuchs, Marc:** *High Performance ATP Systems by Combining Several AI Methods*, to appear in Proc. IJCAI-97, Nagoya, AAAI Press, 1997; also available as SEKI-Report SR-96-09, University of Kaiserslautern, 1996.

[DK96] **Denzinger, J.; Kronenburg, M.:** *Planning for Distributed Theorem Proving: The Teamwork Approach*, Proc. KI-96, Dresden, LNAI 1137, pp. 43-56, 1996.

[DKS97] **Denzinger, J.; Kronenburg, M.; Schulz, S.:** *DISCOUNT. A Distributed and Learning Equational Prover*, Journal of Automated Reasoning 18(2), pp. 189-198, 1997.

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

[DS96a] **Denzinger, J.; Schulz, S.:** *Recording and Analysing Knowledge-Based Distributed Deduction Processes*, Journal of Symbolic Computation Vol. 21, pp. 523-541, 1996.

[DS96b] **Denzinger, J.; Schulz, S.:** *Learning Domain Knowledge to Improve Theorem Proving*, Proc. 13th CADE, LNAI 1104, pp. 62-76, 1996.

[Fu95a] **Fuchs, M.:** *Learning Proof Heuristics by Adapting Parameters*, Proc. 12th ML, Morgan Kaufmann, pp. 235-243, 1995; also available as SEKI-Report SR-95-02, University of Kaiserslautern, 1995.

[Fu96] **Fuchs, M.:** *Experiments in the Heuristic Use of Past Proof Experience*, Proc. 13th CADE, LNAI 1104, pp. 523-537, 1996; also available as SEKI-report SR-95-10, University of Kaiserslautern, 1995.

[HR87] **Hsiang, J.; Rusinowitch, M.:** *On Word Problems in Equational Theories*, Proc. 14th ICALP, Karlsruhe, LNCS 267, pp. 54-71, 1987.

[Hu80] **Huet, G.:** *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, J. of ACM 27, No. 4, pp. 798-821, 1980.

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

[Mc94] **McCune, W.W.:** *OTTER 3.0 Reference manual and Guide*, Tech. rep. ANL-94/6, Argonne National Laboratory, 1994.

[SSY95] **Sutcliffe, G.; Suttner, C.B.; Yemenis, T.:** *The TPTP Problem Library*, Proc. 12th CADE, Nancy, Springer LNAI 814, pp. 252-266, 1994

Appendix: The domain lattice-ordered groups

The examples in this domain investigated in this paper have been included into the TPTP library (see [SSY95]). Here we will present the 6 examples we were not able to prove with DISCOUNT (and also not with Otter).

All six examples have the following axioms in common:

f(f(x,y),z) = f(x,f(y,z))	f(1,x) = x	f(i(x),x) = 1
l(l(x,y),z) = l(x,l(y,z))	l(x,y) = l(y,x)	l(x,x) = x
u(u(x,y),z) = u(x,u(y,z))	u(x,y) = u(y,x)	u(x,x) = x
f(x,l(y,z)) = l(f(x,y),f(x,z))	u(x,l(x,y)) = x	f(l(x,y),z) = l(f(x,z),f(y,z))
f(x,u(y,z)) = u(f(x,y),f(x,z))	l(x,u(x,y)) = x	f(u(x,y),z) = u(f(x,z),f(y,z))

Here f denotes the group operator with the neutral element 1 and the inverse operator i. The lattice is represented by the two binary functions l and u, the greatest **l**ower bound and the least **u**pper bound of two elements.

Then the following axioms and goals have to be added (a,b and c are Skolem constants):

Name	**Additional Axioms**	**Goal**
distrnu		u(a,n(b,c)) = n(u(a,b),u(a,c))
distrun		n(a,u(b,c)) = u(n(a,b),n(a,c))
p08a	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))
p12	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	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))	a = b
p33	n(u(a,i(a)),u(b,i(b))) = 1	f(a,b) = f(b,a)