Experimenting with Machine Learning in Automatic Theorem Proving*

Josef Urban†

January 8, 1999

Abstract

First the notion of the heuristic and its importance for automatic theorem proving is explained. The question of how to construct a useful system of heuristics and notions is posed, and it is argued for the importance of learning. A simple example of learning a lemma is shown using the Inductive Logic Programming system Progol.

Second a general framework for learning heuristics is discussed and the proof checking system Mizar providing a large body of mathematics is found suitable to learn upon. The process of modifying Mizar for use with Progol is shortly described, e.g. turning the mathematical texts to Prolog structures, creating initial notions, creating the initial base of rules from theorems and definitions, preparing proofs for learning, and problems with polymorphism.

Finally first experiments in this environment are described, which include learning the notion of atomic formula and learning how to use initial rules in the proof of Rolle's theorem.

*This work was supported by Esprit Project 20237 Inductive Logic Programming II

†This article is an abridged version of the master thesis done under the guidence of doc. RNDr. Petr Štěpánek, DrSc. at the Faculty of Mathematics and Physics of the Charles University

1 Automatic Theorem Proving and Heuristics

This section should shortly explain the notion of heuristic, its importance for automatic theorem proving (ATP), and it is the motivation for the work described in next sections.

It is well known, that one of the main problems of ATP algorithms is their computational complexity (also combinatorial explosion). In resolution based theorem provers this happens when the empty clause is not derived quickly enough and the prover keeps generating new clauses over and over, until the amount of possible resolutions is intractable.

There are various methods how to guide resolution steps (e.g. which clauses should be taken to be resolved upon), generally called uniform proof procedures (see [Chang and Lee 73]), because they can be used for proving theorems in any axiomatic theory without any domain specific knowledge. Many people (e.g. [Bundy 83]) think that provers based only on resolution are insufficient for proving theorems in more complex areas (e.g. calculus).

One of the methods which try to cope with such problems is looking at the way people solve them and trying to implement human methods in AI algorithms. An example of using this method is Lenat's Artificial Mathematician described in [Lenat 78], where system of heuristics derived from human behavior does mathematical explorations.

Similarly, we can look at the way people prove theorems and try to extract some heuristics for use in ATP. Here by *heuristic* we mean some rule, method or experience which tells us what to do in some situation, it does not need to be perfectly true ... e.g. it suffices that it often helps us, and it does not need to be precisely formulated in some well defined notions, on the contrary, it often uses vague notions (e.g. 'similarity').

Some examples of heuristics used in mathematics are:

in Logic: *'Proving something for all formulas is often done by induction on the complexity of formula'*

in Calculus: *'It has usually little sense to think about the inner structure of a real number as a set'*

in Set Theory: *'Try some diagonalization'*

The system of heuristics we use is probably quite complicated, e.g. sometimes we try to prove something using some method, then we see it is a road to nowhere and think why the previous method failed, and on the basis of it try another method. So we have not only heuristics formulated in the language of its domain theory, but also heuristics formulated by means of other heuristics and results of their application, e.g.

'There is no sense in applying some rule 'R' and immediately after it applying the rule inverse to 'R' '

or

'Try substituting ϵ for δ, and if the result is 2ϵ, try substituting $\epsilon/2$'

In my opinion, the quality of a theorem prover depends immediately on the set of heuristics that its authors recognize, and on the way they link them together and add to some standard proving techniques such as resolution. So it seems we should be interested in the question how to find heuristics, and more generally a sufficiently large system of heuristics, that would give us recommendations for theorem proving similar to those of human.

2 Finding Heuristics, Learning

The most straightforward way of creating heuristics seems to be simply thinking of them and typing them in. It corresponds to teaching done completely by explanation, i.e. we have to provide definition for every notion our heuristics use, define many relations among heuristics, and worse of all, try to find them all on all levels (e.g. heuristics about using heuristics, etc.).

There is probably some redundancy in this method, because there surely are some heuristics about finding heuristics, so why not let them find the rest for us. I think 'learning from examples' could be a (meta)heuristic of this kind and in this work experiments are carried out to find out what is possible to discover using this heuristic.

The reason for trying this heuristic comes again from comparison with humans. When we learn mathematics, it often has the form of watching some theory being developed, i.e. watching some necessary notions being defined, theorems about them being proved and maybe some interesting examples of applying the theory being shown. In this framework there is often very little said about the motivation for defining exactly the notions defined, and leading the proofs of theorems exactly the way it was done. Yet after some time we are able to produce some proofs ourselves or define and explore interesting notions ourselves, we not only remember what was shown to us, but also know heuristics telling us whether something is interesting or not and what to do when we want to prove something.

In my opinion this is (at least partly) caused by our ability to generalise what we are shown to some useful heuristics or vague notions in which we actually think. This generalisation happens often quite spontaneously, e.g. many people when firstly shown the notion of continuous real function would guess it would also mean differentiability, or when shown several proofs in calculus and group theory they would be able to speak about the most striking differences between

the two theories.

So in order to gain some heuristics, we would like to simulate this process of generalising examples in computer. There are various approaches to the task of machine learning and many systems implementing them, the most convenient for the specific task of learning heuristics seemed to me to be the Progol system [Muggleton 95, Roberts 97] coming from the field of Inductive Logic Programming.

3 Progol, Simple Example of Learning a Rule from Examples of Its Use

3.1 Progol

Progol is a Prolog interpreter enhanced by predicates that enable learning. Progol tries to generalise examples given to it to some Prolog clauses whose form we determine by special *mode* predicates, so the method implemented here is called Mode-Directed Inverse Entailment.

The quickest way of showing how to learn something using Progol is probably running through some example, here the example of learning classification of animals which is distributed together with Progol is chosen.

In this example we want to learn definitions of animal classes such as *mammals* or *birds*, from some positive examples such as

class(eagle,bird). class(bat,mammal). class(dog,mammal).

and some background knowledge such as

has_legs(eagle,2). has_legs(bat,2). has_legs(dog,4). has_milk(bat). has_milk(dog).

So we want to learn clauses of the form e.g. *class(A,mammal):- has_milk(A)* . We specify the required form of output clauses by predicates *modeh/2* and *modeb/2*, where we determine which predicates can occur in the head and in the body of output clauses. Here these declarations would look something like

:-modeh(1,class(+animal,#class))?
:-modeb(1,has_legs(+animal,#nat))?
:-modeb(1,has_milk(+animal))?

The first clause says that the output clause should have predicate *class/2* in its head with first term being a variable of type *animal* and the second term being a constant of type *class*. The two following clauses say that in the body we

allow one occurence of the predicate *has_legs/2* and one occurence of the predicate *has_milk/1*. The types used in these declarations have to be defined, here it would be e.g.

animal(eagle). animal(bat). animal(dog). class(bird). class(mammal).

The synbol $'+'$ in the mode declarations specifies we want the term to be a variable (exactly an input variable), while the symbol $'\#'$ means we want the term to be a constant. The third possibility is $'-'$, which represents output variable.

We should also set some parameters influencing the learning process (e.g. how complex hypotheses Progol creates). If everything goes well, Progol really generalizes our examples and produces clauses like

class(A,mammal):-has_milk(A).

3.2 Simple Example of Learning a Rule from Examples of Its Use

The first thing we would like to know when thinking about finding heuristics or interesting notions by machine learning is whether it is possible at all, at least in some simple case.

In the first experiment, a simple lemma (exactly a definition) *sub(x,y)=x+(-y)* was learned from examples where this lemma was used and some counterexamples. Here the simplification is that

- we know beforehand what we learn and can supply additional examples or counterexamples as well as adjust parameters for learning to obtain the right result
- the learned lemma is not a heuristic defined in possibly vague notions, we can constrain the search to mathematical expressions
- the background knowledge used in the learning did not go too much in detail, thus the learned clauses would have some drawbacks.

The positive examples for this learning were produced using the IMPS (Interactive Mathematical Proving System). Several mathematical expressions containing the term *sub(...,...)* (e.g. $sub(x*z, z*x) = 0$) were typed in and IMPS rewrote them using our lemma, so the rewritten expressions would instead of the term *sub(...,...)* contain terms like *(...)+(-(...))* (e.g. $(x*z) + (-(z*x)) = 0$).

The pairs $[expression\_before\_rewriting, expression\_after\_rewriting]$ were given to Progol as examples of the predicate *similar/2* and some counterexamples were

added. We would like Progol to find what all these examples have in common. That's why the predicate *gener/2* was defined, where *gener(A,B)* should stand for *'A unifies (in the sense of IMPS) with some term in B'*. Following previous example,

$$gener(sub(x, y), sub(x * z, z * x) = 0)$$

would be true, because $sub(x, y)$ unifies with the term $sub(x * z, z * x)$ [1]. Given these examples and definitions, we can use mode declarations to specify the exact form of the clause searched for:

modeh(1,similar(+expr,+expr))?
modeb(2,gener(#expr,+expr))?

This says in the head there should be something like *similar(A,B)*, while in the body we allow two occurences of the predicate *gener/2* each of them having a constant term as a first argument. What we hope for is that the outcome will be the clause

$$similar(A, B) : -gener(sub(x, y), A), gener(x + (-y), B).$$

which means in all examples the *expression_before_rewriting* contains a term generalised by $sub(x, y)$ and the *exprression_after_rewriting* contains a term generalised by $x + (-y)$, and in which the learned lemma is already apparent.

Of course, we could impose some stronger conditions, e.g. we could require in each pair that the generalisation in both expressions occurs at the same position. I think it is not necessary, because when there is a lot of examples and counterexamples then there is a good chance that the wrong hypotheses will not succeed in generalising them.

The real simplification that took place in defining predicate *gener/2* was that only one 'generalising variable' (e.g. 'x') was allowed. You can genaralise any term by a variable, but when learning, this variable will always be 'x'. I think it would be possible to correct this drawback, but it would be necessary to remember somehow that 'x' is already bound to some term, and always check whether a new term is already generalised by some variable.

So the clause Progol actually learned differs from above in that instead of two variables 'x','y', it contains only one variable 'x':

$$similar(A, B) : -gener(sub(x, x), A), gener(x + (-x), B).$$

This is what it really looks like when the λ-calculus notation of IMPS turned to lists is used:

[1] The notation here is simplified in that IMPS uses λ-calculus and IMPS expressions were turned to lists for use in Progol.

```
similar(A,B) :-
        gener([[apply-operator,[sub,[rr,rr,rr]],[x,rr],[x,rr]],rr],A),
        gener([[apply-operator,[[lambda,[[rr,x,y]],[[apply-operator,
                [+,[rr,rr,rr]],[[apply-operator,[-,[rr,rr]],[y,rr]],rr],
                [x,rr]],rr]],[rr,rr,rr]],[x,rr],[x,rr]],rr],B).
```

This learning used five examples and two counterexamples and took about 4 seconds on P133.

4 A Framework for Larger Experiments

In my opinion the previous example shows that in principle it is possible to find in mathematical texts some knowledge that is not explicitly presented there, it is implicit or hidden (... sort of kabbalah).

Finding some nontrivial and useful heuristics in some larger text is certainly going to be harder, because

- we will have to learn how to choose from a larger text some parts that are suspect to have something in common (so that we could do some generalisation on them)
- the heuristics we know use a language that differs from that of mathematics, e.g. it contains notions like similarity, inverse rule, induction, triviality, heuristic, integral, theorem of calculus, etc, and in the beginning, we do not know definitions of these notions.

I think that the two problems of finding heuristics and finding notions that heuristics use should be solved in the same time. It may happen, that some conditions will occur quite often in the definitions of the heuristics found so far, e.g. it may turn out that some group of lemmas is used to some expression only when it contains some typical functions or predicates. We can then give some name to this kind of expression, e.g. *'integral'*, or *'expression of set theory'*, store their definition in lower notions somewhere (e.g. *'predicate_symbols(X,[ϵ])'* for X being an *'expression of set theory'*), and use them for finding more complex heuristics.

So when we want to try finding heuristics and their language, we should have

- some mathematical texts, where we will search for the heuristics and their notions
- some basic notions that describe these texts (e.g. knowing that $'\epsilon'$ is a predicate symbol comes in handy)
- some system that should enable us to choose and prepare parts of the mathematical texts for learning

- a learning system ... here it is Progol

So far we often used comparison with human thinking as a method telling us what to do next. If we want to continue using this method, the mathematical texts we learn upon should be as similar to human textbooks as possible. On the other hand, a genuine textbooks already contain great deal of natural language and vague notions and it would be hard even to construct some set of initial notions describing the structure of the texts (e.g. telling the computer what is a theorem or proof or even a correct proof).

It follows that we are looking for mathematical texts that are formalised in some way, but at the same time they are as close to human mathematics as possible. I finally chose the MIZAR system, which will be now shortly described.

5 The MIZAR system

The MIZAR system [Rudnicky 92] is an outcome of a project started in Poland in 1973. In this project a syntax for writing mathematical articles was developed and many articles were written using this syntax. The syntax was created to comply with two major requirements:

- it should make computer checking of the correctness of articles possible
- it should resemble human mathematics as much as possible, thus making it easier for the mathematicians to write their articles in this form

So the distribution of the MIZAR system (found on internet) consists of two parts, the system checking the correctness of newly written articles and about 500 articles from various fields of mathematics that were already checked and approved to be in the distribution. The articles not only have some informative value, they also enable us to use notions defined or theorems proved in them, thus making writing new articles in more complex fields quite easy.

Here some parts of the article about sets are shown:

```
:: Some Basic Properties of Sets
::  by Czes{\l}aw Byli\'nski

environ
 vocabulary BOOLE,FAM_OP;
 theorems TARSKI,BOOLE,ENUMSET1;

begin
  reserve v,x,x1,x2,y,y1,y2,z for Any;

_Th12:
    {x} c= X iff x ∈ X
  proof
```

```
   thus {x} c= X implies x ∈ X
    proof x ∈ {x} by TARSKI:def 1; hence thesis; end;
   assume
A1:  x ∈ X;
   let y;
   thus thesis by A1,TARSKI:def 1;
  end;

definition let X1,X2;
 func [:  X1,X2 :]  means
:Def1:  z ∈ it iff ex x,y st x ∈ X1 & y ∈ X2 & z = [x,y];
....
```

Articles consist of keywords, e.g. *environ, reserve, theorem, proof, definition, thus, hence* , comments, initial declarations (names of articles we will refer to), reservation of variables for certain types (here the type *Any* means *set*), theorems, definitions, schemes and auxiliary items. Auxiliary items often have the same form as theorems, _Th12 is of this kind, it is some lemma with proof. The definition shown is the beginning of the definition of cartesian product, after the defining formula there should be some proofs of correctness of this definition. The structure of schemes is similar to that of theorems, but they usually contain some quantification of a predicate or function of some type.

The meaning of keywords is quite similar to their meaning in English and the syntax allows us to write the proofs in quite a common way, e.g. it is possible to prove some part by contradiction or to introduce in the course of proof some lemma and give a proof of it. Each article can define its own notation, so one symbol can have different meanings in differrent articles, and even in one article a symbol can have different meanings, e.g. when it is a function or predicate defined in a differrent way for various types.

6 Preparing MIZAR for Learning

It was necessary for the task of learning to make MIZAR articles at least to some extent understandable for Progol. Firstly some initial notions describing the articles were created and then some more complicated predicates were written and initial base of rules created, which made proof analysis possible.

6.1 Creating Initial Notions, Turning MIZAR Texts to Prolog Structures

Here the syntax of the MIZAR language in definite clauses found on the home page of the MIZAR project was of great help. Definite clauses such as

Atomic-Formula-Expression =
Term-Expression-List Predicate-Symbol Term-Expression-List

were turned to Prolog predicates about recursive lists such as

```
atomic_formula_expression([A,B,C]) :-
    term_expression_list(A), predicate_symbol(B), term_expression_list(C).
```

Similar predicates working basically on the principle of differential lists were used for parsing the linear text of articles to recursive lists, e.g. here it would look like

```
atomic_formula_expression(V0,V,[A,B,C]) :-
    term_expression_list(V0,V1,A), predicate_symbol(V1,V2,B), term_expression_list(V2,V,C).
```

In fact, this process was quite complicated, because using only the method of differential lists would often be very inefficient and parsing the terms and formulas requires some additional knowledge (e.g. priorities of symbols), that can be for each article different.

6.2 Problems with Polymorphism

In MIZAR there are many types, and many functions and predicates have different definitions for different types, and sometimes it is quite complicated to tell exactly which definition was used (e.g. function defined for some type may be used to a more special type, however when definition for the more special type exists, it should be used, etc).

This seemes to be quite unsuitable for some possible learning in Progol, because only to be sure we understand some term or formula correctly, we would have to include all that polymorphism resolving machinery in the background knowledge, slowing down the actual learning.

So an attempt was made to get rid of the polymorphism, basically by numbering all different meanings of functions, predicates and types, and preprocessing the articles with a program which tries to guess the right meanings and adds the corresponding numbers to symbols for functions, predicates and types.

6.3 Creating Initial Rules

If we want to analyse proofs, we should be able to apply some lemma or theorem in some proof situation. That is why applicable rules were constructed from the theorems, definitions and schemes in MIZAR articles.

It is easy to create a rule from a theorem of the form *'A implies B'*. It already is a rule, applicable in case we want to prove an instance of *'B'*, changing it to an instance of *'A'*. More generally, when we want to prove a disjunction $\bigvee\{A_1, \ldots, A_n\}$ and we know that disjunction $\bigvee\{B_1, \ldots, B_m\}$ is true, the algorithm is:

1. find the intersection[2] $\{C_1, \ldots, C_l\} = \{A_1, \ldots, A_n\} \cap \{B_1, \ldots, B_m\}$
2. if this intersection is empty, then $\bigvee\{B_1, \ldots, B_m\}$ is not applicable
3. else let $\{D_1, \ldots, D_{m-l}\} = \{B_1, \ldots, B_m\} \setminus \{C_1, \ldots, C_l\}$, the proof of $\bigvee\{A_1, \ldots, A_n\}$ is reduced to the proof of $\bigwedge\{\neg D_1, \ldots, \neg D_{m-l}\}$

for if none of D_i is true, some C_j must hold (because $\bigvee\{B_1, \ldots, B_m\}$ is true) and consequently $\bigvee\{A_1, \ldots, A_n\}$ holds.

So theorems from MIZAR articles were converted to conjunction of disjunctions, and each disjunction was taken to be an initial rule (it means usually more than one rule were created from a theorem). Basically the same was done with schemes and definitions, definitions first being turned to equivalences.

The problem here arises if existential quantification is present in some theorem, scheme or definition. MIZAR does not require some skolemization, in the course of proof special *'reasoning items'* are used to handle existential quantification. That is why the parts of formulas beginning with existential quantifier were left intact, and the plan is to use similar *reasoning items* for their eliminating as MIZAR does.

6.4 Preparing Proofs for Learning

The MIZAR proofs consist of various *reasoning items*. These items in most cases modify the *thesis*. At the start of a proof the *thesis* is simply the theorem being proved, and the proof is finished when there is nothing else to prove, i.e. the *thesis* is empty.

The problem is, that the current *thesis* is apparent only at the beginning and at the end of the proof, and if we would like to know the *thesis* somewhere in the middle of the proof, we would have to run through all *reasoning items* from the beginning to that point and apply them. It is quite inconvenient, because we could want to learn e.g. why exactly some *reasoning item* was used at some point of the proof, and for such learning knowing the form of the *thesis* at that point is essential.

That is why a program which determines the *thesis* was written. Proofs are before learning preprocessed with this program, and it usually adds the current *thesis* to *reasoning items* and also numbers the *reasoning items* in a uniform way, to facilitate orientation in the proof.

[2] more exactly: intersection up to an unification; this applies also to step 3.

7 First Experiments with Learning

7.1 Learning the Notion of Atomic Formula

This experiment was done to try, whether under very favourable conditions it is possible at all to learn some useful notion. It was carried out in an early stage of preparing MIZAR texts for use with Progol, also with the purpose to get feedback for further modification of MIZAR texts.

For this learning, the notion of *'atomic formula'* was chosen, and the task was to learn this notion using some other initial notions (e.g. *'term', 'identifier'*, etc) and some examples of this notion. The simplification here was:

- We know beforehand the notion we learn, only examples of this notion were supplied (together with some counterexamples)
- About 100 initial notions describing the texts were created from the MIZAR syntax. It would be intractable to allow them all in the learning. On the other hand, we have no heuristics yet, that could do the task of choosing the most 'suspect' initial notions, that should be allowed in the learning. That is why I chose 8 'suspect' initial notions myself, and of course, among them there were also the notions necessary for the definition of *'atomic formula'* (i.e. *term_list* and *predicate*)

The examples of atomic formulas were created by choosing all atomic formulas in the proof of Rolle's theorem. There were about sixty of them, one counterexample was added. In this early stage of modifying MIZAR articles, differential lists were used instead of recursive lists, i.e. an example of atomic formula would look like

```
f_e([f1,'.',x,'=',f2,'.',x,'+',r],[])
```

where *f_e/2* stands for *formula expression* (although it should be *atomic formula expression*), and *f_e(A,B)* means *'append(C,B,A) holds, and C is an atomic formula'*. In all these examples the second argument was the empty list, so the whole first argument is the atomic formula.

We want to learn the definition of the predicate *f_e/2*, therefore the head mode declaration is

```
:-modeh(1,f_e(+list,[]))?
```

The differential lists are quite unsuitable for working with recursive things such as formulas or terms (it is very inefficient), on the other hand here this formalisation makes writing the body mode declarations quite easy. here they have the form

```
:-modeb(3,term_expression(+list,-list))?
:-modeb(3,identifier(+list,-list))?
:-modeb(3,predicate_symbol(+list,-list))?
....
```

It means Progol will construct hypotheses such as

$$f\_e(A,[]) :- term\_expression(A,B), identifier(A,B), predicate\_symbol(B,C).$$

and it already is the kind of definition we are looking for.

The number 3 in the body mode declarations is quite arbitrary, it says that the output clause should not contain more than three occurences of each of this predicates. The more occurences we allow, the more complex will the search be, so it is an attempt to balance the speed of the search with its generality.

The final learning used about 60 positive examples and 1 counterexample, and on PC486/100MHz took (depending on parameter settings) from 50 to 100 seconds. It is very much, mainly because handling terms and formulas as differential lists was inefficient. All examples were generalised by the clause

```
f_e(A,[]):- term_expression(A,B), predicate(B,C), term_expression(C,D).
```

7.2 Learning How to Use Initial Rules in the Proof of Rolle's Theorem

One particular approach to learning heuristics is to learn on various *proof situations* (described by the formula being proved, the special symbols in it, the theory to which the formula belongs, etc.), the *set of possible solutions* of these situations (i.e. the initial rules applicable to the formula being proved, or more genarally the heuristics applicable to the proof situation) and the *solution actually used* in the proof.

To put it more concretly, we could create from the MIZAR proofs triples

(1) [*proof situation, its possible solutions, the solution actually used in the proof*]

and try to generalise them.

In this setting, some of the heuristics already mentioned could be (at least in principle) learnable, e.g. we could find out that if the *proof situation* has attributes *'model theory'* and *'st. is being proved for the set of all formulas'*, then *'induction on the complexity of formula'* is always in the *set of possible solutions* and quite often it is also the *solution actually used in the proof.*

For the first learning in this setting, the situation was simplified in that

- as the *set of possible solutions* only the initial rules applicable to the formula were taken (we do not know any higher heuristics yet)
- the description of proof situation was ommited. It means the examples for learning will be only pairs

  (2) [*applicable initial rules,the rule actually used in the proof*]

In fact, the set of applicable rules describes the proof situation to some extent, so the first two members in the triples (1) are in a simple way described by the first member in the pairs (2).

The examples for this learning were obtained from the proof of Rolle's theorem (theorem 'Th1' in the MIZAR article 'rolle.miz'). The *reasoning items* of this proof (there are hundreds of them) were first preprocessed in the way described in (6.4.), and from them only the items that have the form of self-contained statements were chosen. Examples of such statements are

```
A0:  p ≤ g by Z0;
A01:  ].p,g.[ is open by RCOMP_1:25,Z0;
r/(n+2)<r by SQUARE_1:14;
C9:  rng (h1+c) c= N1 proof ...  end;
```

These items can be divided into three groups:

- items with a longer proof beginning with the keyword *'proof'* and ended with the keyword *'end'* ... the item C9 above
- items with a *'simple justification'* (it is a list of references to other items, theorems, schemes or definitions, from which the current item immediatelly follows), that contain a reference to some other item in this proof ... items A0 and A01 above ... they contain reference to the item Z0, which is some assumption made previously in the proof
- items with a *'simple justification'* that contain only references to other theorems, schemes or definitions ... the third item from above ... contains reference to theorem 14 in the article 'SQUARE_1.miz'

The simplest possibility was to choose only the third group for learning, where still remains 54 items. Thus we may take into account only the theorems, schemes and definitions previously proved, while if we wanted to use also the items from the first and second group, we would have to consider also references to other items or assumptions made in the proof, and the learning would have to be more complicated.

There are 55 MIZAR theorems, definitions and schemes used in the proof of this theorem (see Appendix A), and from them 90 initial rules were constructed. For the learning only these 90 rules were allowed. Other possibility is e.g. to allow all rules constructed or to allow all rules from the articles declared in the environment declarations of the article 'rolle.miz'. But with both these possibilities the learning would take very long time, because there are about 7000 initial rules (made from about 60 articles that were necessary for the article 'rolle.miz') and there are still thousands of rules in articles directly referenced in 'rolle.miz'.

To create examples for learning, it was necessary for each of the 54 chosen items to determine the set of applicable rules. It was done using the program described in 6.3. The statistics of the results is in Appendix B (this is no learning yet). In this statistics, two things should be noticed and explained:

- Sometimes according to the original proof some rule should be used, but my program for rule application said the rule was not applicable. This is usually caused by the incompletness of the program for rule application (e.g. the existential quantifiers are not handled properly yet).
- The number of possible applications of some rules is very high (thousands). It happened because the rules of the form *'a=b'* were also allowed to be used for rewriting the left hand side to the right hand side of the equation and vice versa. In case one side of the equation is a variable, it unifies with almost anything, and the rule is applicable many times in different ways.

Besides this statistics, the main result of running the program determining the applicability of rules, are the 54 pairs [*list of applicable rules, the rule actually used in the proof*] (one for each of the chosen reasoning items). For example, the pair

```
[[RCOMP_1:15,SQUARE:14,BOOLE:def8],RCOMP_1:15]
```

means that to prove one of the 54 reasoning items the theorem *RCOMP_1:15* was used, but theorem *SQUARE:14* and definition *BOOLE:def8* were also applicable.

In addition to these 54 positive examples, about 400 counterexamples were created, taking the 54 lists of applicable rules (the left hand sides of the pairs) and for each list, all its members (except those in the positive examples) were taken as a right hand side of the counterexample pairs. It means for the example above we would create two counterexamples

```
[[RCOMP_1:15,SQUARE:14,BOOLE:def8],SQUARE:14]
[[RCOMP_1:15,SQUARE:14,BOOLE:def8],BOOLE:def8]
```

In this first experiment, positive examples (denoted e.g. by predicate *choose/2*) were generalised using the predicates *p_element/2, n_p_element/2* which have for the initial rules the same meaning as *member/2, not member/2*. Two learnings were carried out, in the first one the posistive and negative examples were as already described, while in the second one the positive and negative examples were exchanged (i.e. there were about 400 positive examples and 54 counterexamples).

In both cases the mode declarations are

```
:-modeh(1,choose(+rule_list,#rule))?
:-modeb(20,p_element(#rule,+rule_list))?
:-modeb(2,n_p_element(#rule,+rule_list))?
```

So we will learn clauses of the form

```
choose(A,SEQ_4:46) :  - p_element(A,SEQ_1:def8).
```

which in the first case means:

'If both `SEQ_4:46, SEQ_1:def8` *are applicable, then the rule* `SEQ_4:46` *should be used'*

while in the second case the meaning is:

'If both `SEQ_4:46, SEQ_1:def8` *are applicable, then the rule* `SEQ_4:46` *should not be used'*

7.2.1 Results of This Learning

In the first case the learning was quite unsuccesful, while in the second case the results seem to be quite good. I think the reason is the ratio of positive and negative examples.

In the first case the number of counterexamples was too high to allow good generalisation (i.e. the closed world assumption used when they were created was too strong). The 54 positive examples were generalised to 50 clauses, from which only 18 were newly created definitions, while the rest were the original examples.

The second case is more interesting, the about 400 positive examples were generalised to 140 clauses, only 34 of them were the original examples, while the rest were newly created definitions. This learning took about 2,5 hour on PC486/100MHz.

More than 100 newly created definitions are of the form

```
choose(A,REAL_1:33) :- p_element(NAT_1:38,A).
```

which in this case means that if both these rules are applicable, `REAL_1:33` should not be used. This form of the most of the newly created clauses tempts to think of them as of defining something very similar to ordering of the initial rules used in the proof (e.g. the previous clause would say `REAL_1:33 < NAT_1:38`). I tried to analyse a little this "ordering", to find out something about the results of this learning.

For 24 rules[3] of the 55 holds, that there is no "better" rule, which could be (with some generalisation) interpreted as a hint to use these rules whenever they are applicable. As early as we look at the first rule of these 24 ... `AXIOMS:13`

[3]In Appendix A there is '+' before them

... associtivity of '+', we realize, that either the learning set was too special or our interpretation of the results is too general, because using associativity of '+' whenever it is possible is surely not the right way to prove theorems.

On the other hand, all of the three rules that are atomic formulas and do not contain predicate '=' (i.e. `NAT_1:18, NAT_1:21, RFUNCT_2:10`) are among the best 24. I think it corresponds to the fact that atomic formulas not containing '=' have in proving the meaning of 'facts', used in the backchaining whenever possible (atomic formulas containing '=' can also be used for rewriting, that is why most of them (7 of 8) are not among the 24).

Another quite expectable fact is that many of the "top 24" contain some unusual (in relation to other rules) predicate or group of terms (e.g. *'diff(f,x0)', 'f(Y)', 'is_constant_on', '0≤y-x'*, etc.).

On the other side of the scale there are six rules, that according to our interpretation should be used only when no other rule is applicable. They are

`FDIFF_1:def8, REAL_1:26, REAL_1:33, SEQ_2:2, SQUARE_1:14, TARSKI:def2`

With the exception of `REAL_1:24` all of them contain some expresion of the form *'x=...'*, the 'most applicable' (according to statistics given in Appendix B) rule `TARSKI:def2` even containing two expressions of the form *'x=y'*.

Finally I created for each rule the list of 'better' rules. Here the 5 cases where this list was of length 1 (i.e. the 'second best' rules) were interesting. These pairs (the second best, the better) are:

```
  NAT_1:19   AXIOMS:13
RCOMP_1:11   RCOMP_1:6
RCOMP_1:15   BOOLE:8
   REAL:39   SEQ_2:6
 REAL_1:53   REAL_1:14
```

The rules in the first pair have nothing in common, those in the second pair both contain the term *'[.g,s.]'*, the rules in the third pair both contain the predicate '∈', and the rule BOOLE:8 that contains more general terms at this predicate is 'better', in the fourth pair both rules contain the term *'x/y'* and atomic formula not containing '=' is prefered, and in the last pair the rules contain terms $x+y$, and the rule REAL_1:14 with the more special term *'x+ -y'* is prefered.

With the exception of the first pair, the rules in these pairs are somewhat similar, they contain the same predicate or similar group of terms. It leads to the conjecture, that rules which are close in the 'ordering' found, are often close also syntactically and applicable in similar situations.

8 Conclusions

The purpose of this work is to try to develop a system for finding useful notions and heuristics for theorem proving. The method that is tried is to analyse the

mathematical texts of the MIZAR system using the initial notions of MIZAR syntax, the initial rules created from the MIZAR theorems and definitions, some simple theorem proving predicates written in Prolog, and the Inductive Logic Programming system Progol.

Three Progol learnings were described. The results of the first (learning a rule from examples of its use) and the second one (learning the notion of atomic formula) were previously known, they were carried out to check the feasibility of this approach at least under very favourable conditions. The result of the third and largest learning (how to use initial rules in the proof of Rolle's theorem) is a compression of about 400 examples taken from the proof to 140 clauses, which when viewed as defining ordering of the initial rules give already some meaningful heuristics, e.g. *'Choose atomic formula not containing '=' whenever it is possible'.*

9 Further Work

We could study the 'ordering' further, but there is the danger that we will try to find some deep relations in results of quite a superficial learning. The plan is rather to try some less trivial learnings, e.g. to add some other attributes of proof situations and to include the heuristics found so far (here it would be the 'ordering' found) to the set of possible solutions, and in longer term to try automating the process of succesive learnings to create some hierarchies of heuristics.

10 Acknowledgements

I would like to thank doc. RNDr. Petr Štěpánek, DrSc. for advice and help with this work, and also doc. RNDr. Olga Štěpánková, CSc., from the Czech Technical University for providing contact to the Esprit Project 20237 Inductive Logic Programming II. Many thanks to people working on Progol and on the MIZAR project for making their software publicly available.

References

[1] [Bundy 83] Bundy, A. The Computer Modelling of Mathematical Reasoning. Academic Press, 1983.

[2] [Chang and Lee 73] Chang C-L. a Lee R. C-T. Symbolic Logic and Mechanical Theorem Proving. Academic Press, 1973.

[3] [Jirků a kol. 91] Jirků, P., Štěpánek, P., Štěpánková, O. Programování v jazyku Prolog. SNTL - Nakladatelství technické literatury, 1991.

[4] [Lenat 78] Lenat, D.B. The Ubiquity of Discovery. Artificial Intelligence 9: 257-285, 1978.

[5] [Muggleton 95] Muggleton, S. Inverse Entailment and Progol. New Generation computing, 13 (1995), p. 245-286.

[6] [Roberts 97] Roberts, S. An Introduction to Progol. Article from Progol home page, 1997, ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/progol4.2/manual.ps

[7] [Rudnicky 92] Rudnicky, P. An Overview of the MIZAR Project. Article from MIZAR home page, 1992, ftp://ftp.cs.ualberta.ca/pub2/Mizar/httpd/htdocs

Appendix A

The theorems and definitions used in the proof of Rolle's theorem.

```
+AXIOMS 13        x+(y+z)=(x+y)+z.
AXIOMS 22         x ≤ y & y ≤ z implies x ≤ z;
+BOOLE 29         X c= Y & Y c= Z implies X c= Z;
BOOLE 8           x ∈ X U Y iff x ∈ X or x ∈ Y;
+FCONT_1 30       Y c= dom f & Y is compact & f is_continuous_on Y implies (f(Y)) is compact;
FCONT_1 def 2     pred f is_continuous_on X means X c= dom f & for x0 st x0 ∈ X holds f|X
                  is_continuous_in x0;
+FDIFF_1 16       f is_differentiable_on Z iff Z c=dom f & for x st x ∈ Z holds f
                  is_differentiable_in x;
+FDIFF_1 20       for f,x0 for N being Neighbourhood of x0 st f is_differentiable_in x0
                  & N c= dom f holds for h,c st rng c = {x0} & rng (h+c) c= N holds
                  h"_(f.(h+c) - f.c) is convergent & diff(f,x0) = lim (h"_(f.(h+c) - f.c));
FDIFF_1 30        Z c= dom f & f is_constant_on Z implies f is_differentiable_on Z & for x st
                  x ∈ Z holds (f'3Z).x =0;
FDIFF_1 def 8     func f'3X -> PartFunc of REAL,REAL means dom it = X & for x st x ∈ X holds
                  it.x = diff(f,x);
+FUNCT_1 102      y ∈ foX iff ex x st x ∈ dom f & x ∈ X & y = f.x;
+NAT_1 18         for k holds 0 ≤ k;
NAT_1 19          0 <> k implies 0 < k;
+NAT_1 21         0 <> k + 1;
+NAT_1 38         k < n + 1 iff k ≤ n;
+PARTFUN2 57      f is_constant_on X & Y c= X implies f is_constant_on Y;
RCOMP_1 11        for g,s st g≤s holds [.g,s.]  = ].g,s.[ U {g,s};
RCOMP_1 15        (p<g implies ].p,g.[<>∅) & (p≤g implies p∈[.p,g.]
                  & g∈[.p,g.]  & [.p,g.]<>∅ & ].p,g.[ c= [.p,g.]);
+RCOMP_1 24       for s,g st s≤g holds [.s,g.]  is compact;
+RCOMP_1 25       for p,q st p<q holds ].p,q.[ is open;
+RCOMP_1 28       (ex r st r∈X) & X is compact implies X is bounded;
RCOMP_1 6         [.g,s.]  = {r :  g≤r & r≤s };
RCOMP_1 7         ].g,s.[ = {r :  g<r & r<s };
REAL_1 14         x-y=x+ -y;
REAL_1 26         -0=0;
REAL_1 33         x<>0 implies (1/x=x" & 1/x"=x);
REAL_1 39         y<>0 implies (-x/y=(-x)/y & x/(-y)=-x/y);
REAL_1 50         x≤y iff -y≤-x;
REAL_1 53         x ≤ y iff x+z≤y+z;
REAL_1 59         x<y implies x+z<y+z & x-z<y-z;
REAL_1 66         x < 0 iff 0 < -x;
+REAL_1 67        ((x<y & z≤t) or (x≤y & z<t) or (x<y & z<t)) implies x+z<y+t;
+REAL_1 72        0<z implies 0<z";
REAL_1 73         0<z implies (x<y iff x/z<y/z);
REAL_1 83         -(x-y)=y-x;
+REAL_1 92        ((x ≤ y & z ≤ t) implies x - t ≤ y - z) & (((x < y &
                  z ≤ t) or (x ≤ y & z < t) or (x < y & z < t)) implies x-t < y-z);
+RFUNCT_2 10      seq.n ∈ rng seq;
RFUNCT_2 18       seq is convergent & (for n holds seq.n≤0) implies lim seq ≤ 0;
+RFUNCT_2 19      (for n holds seq.n∈Y) implies rng seq c= Y;
```

```
+RFUNCT_2 42      Y c= dom h & hoY is bounded & upper_bound (h(Y)) = lower_bound (h(Y))
                  implies h is_constant_on Y;
SEQ_1 7           seq is_not_0 iff for n holds seq.n<>0;
+SEQ_1 def 8      func seq" -> Real_Sequence means for n holds it.n=(seq.n)";
+SEQ_2 10         0<r1 & r1<r & 0<g implies g/r<g/r1;
SEQ_2 2           g/2 +g/2=g & g/4 +g/4=g/2;
SEQ_2 31          seq is convergent & (for n holds 0≤seq.n) implies 0≤(lim seq);
+SEQ_2 6          0<g & 0<p implies 0<g/p;
SEQ_4 24          (X is bounded & ex r st r∈X ) implies (lower_bound X)≤(upper_bound X);
SEQ_4 46          0<r & (for n holds seq.n=g/(n+r)) implies seq is convergent & lim seq=0;
+SEQ_4 def 3      attr X is bounded means X is bounded_below & X is bounded_above;
SEQ_4 def 4       func upper_bound X ->Real means(for r st r∈X holds r≤it)
                  & (for s st 0<s ex r st(r∈X & it-s<r));
SEQ_4 def 5       func lower_bound X ->Real means (for r st r∈X holds it≤r)
                  & (for s st 0<s ex r st (r∈X & r<it+s));
+SQUARE_1 12      x ≤ y implies 0 ≤ y - x;
SQUARE_1 14       x/1 = x;
TARSKI def 2      func { y, z } -> set means x ∈ it iff x = y or x = z;
ZFMISC_1 37       {x} c= X iff x ∈ X;
```

Appendix B

The statistics of usage of theorems and definitions in the proof of Rolle's theorem.

The first number says how many times the rule was really used in the original proof, the second number says how many times it was possible to apply this rule (using the program for rule application I created), and the third is the number of situations in which both ocurred simultaneously, i.e. according to the original proof the rule should be used, and at the same time according to my program it was applicable at that place. If everything were ideal, the first and the third numbers should be the same.

```
      [['RCOMP_1',':',['25']],                    1,2,1].
[['FCONT_1',':',[['def','2']]]                    1,12,1].
       [['RCOMP_1',':',['24']]                    1,1,1].
       [['RCOMP_1',':',['15']]                    2,4,2].
        [['RCOMP_1',':',['6']]                    1,10,1].
       [['FCONT_1',':',['30']]                    1,1,1].
       [['RCOMP_1',':',['28']]                    1,1,1].
  [['SEQ_4',':',[['def','3']]]                    1,3,2].
         [['SEQ_4',':',['24']]                    1,0,0].
      [['RFUNCT_2',':',['42']]                    1,2,1].
     [['PARTFUN2',':',['57']]                     1,2,1].
       [['REAL_1',':',['73']]                     1,143,2].
       [['REAL_1',':',['67']]                     2,46,6].
         [['SEQ_2',':',['2']]                     2,1072,20].
       [['RCOMP_1',':',['7']]                     1,9,0].
      [['FDIFF_1',':',['30']]                     1,362,3].
[['FDIFF_1',':',[['def','8']]]                    1,950,13].
       [['RCOMP_1',':',['11']]                    1,12,2].
         [['BOOLE',':',['8']]                     1,48,4].
 [['TARSKI',':',[['def','2']]]                    1,3824,40].
  [['SEQ_4',':',[['def','5']]]                    2,0,0].
      [['FDIFF_1',':',['16']]                     1,3,1].
        [['NAT_1',':',['19']]                     8,20,8].
         [['SEQ_1',':',['7']]                     4,0,0].
        [['NAT_1',':',['21']]                     10,20,10].
        [['SEQ_4',':',['46']]                     4,380,16].
        [['NAT_1',':',['18']]                     4,8,4].
       [['REAL_1',':',['53']]                     4,56,20].
        [['NAT_1',':',['38']]                     4,28,4].
       [['AXIOMS',':',['13']]                     14,28,14].
        [['SEQ_2',':',['10']]                     4,9,8].
     [['SQUARE_1',':',['14']]                     4,1068,28].
       [['REAL_1',':',['92']]                     2,16,12].
         [['SEQ_2',':',['6']]                     12,22,20].
       [['AXIOMS',':',['22']]                     8,52,0].
       [['REAL_1',':',['59']]                     6,3,0].
       [['REAL_1',':',['14']]                     4,34,6].
      [['FDIFF_1',':',['20']]                     4,21,16].
```

```
    [['RFUNCT_2',':',['18']]                    2,0,0].
    [['FUNCT_1',':',['102']]                    5,13,5].
[['SEQ_4',':',[['def','4']]]                    2,0,0].
    [['SQUARE_1',':',['12']]                    4,8,8].
      [['REAL_1',':',['83']]                    2,32,4].
[['SEQ_1',':',[['def','8']]]                    4,18,12].
      [['REAL_1',':',['26']]                    14,360,20].
      [['REAL_1',':',['50']]                    14,52,4].
      [['REAL_1',':',['72']]                    4,2,2].
      [['REAL_1',':',['39']]                    2,46,4].
      [['REAL_1',':',['33']]                    4,1016,20].
      [['REAL_1',':',['66']]                    2,36,0].
    [['RFUNCT_2',':',['10']]                    10,8,8].
    [['RFUNCT_2',':',['19']]                    4,12,4].
       [['BOOLE',':',['29']]                    7,17,7].
    [['ZFMISC_1',':',['37']]                    4,22,0].
       [['SEQ_2',':',['31']]                    2,0,0].
```