Automated Proof Compression by Invention of New Definitions

Jiří Vyskočil[1]*, David Stanovský[2]†, and Josef Urban[3]‡

[1] Czech Technical University in Prague
[2] Charles University in Prague
[3] Radboud University, Nijmegen

Abstract

State-of-the-art automated theorem provers (ATPs) are today able to solve relatively complicated mathematical problems. But as ATPs become stronger and more used by mathematicians, the length and human unreadability of the automatically found proofs become a serious problem for the ATP users. One remedy is automated proof compression by invention of new definitions.

We propose a new algorithm for automated compression of arbitrary sets of terms (like mathematical proofs) by invention of new definitions, using a heuristics based on substitution trees. The algorithm has been implemented and tested on a number of automatically found proofs. The results of the tests are included.

1 Introduction, motivation, and related work

State-of-the-art automated theorem provers (ATPs) are today able to solve relatively complicated mathematical problems [McC97], [PS08], and are becoming a standard part of interactive theorem provers and verification tools [MP08], [Urb08]. But as ATPs become stronger and more used by mathematicians, understanding and refactoring the automatically found proofs becomes more and more important.

There is a number of examples, and significant amount of more or less successful relevant work in the field of formal proof refactoring. The most well-known example is the proof of the Robbins conjecture found automatically by EQP. This proof has been semi-automatically simplified by the ILF system [Dah98], and later also rewritten as a Mizar formalization [Gra01]. Other examples include the refactoring of the proof of the Four Color Theorem by Gonthier [Gon07], the hint strategy used regularly to simplify the proofs found automatically by the Prover9 system [Ver01], translation of resolution proofs into assertion level natural deduction proofs [Hua96], various utilities for formal proof refactoring in the Mizar system, and visualization of proofs and their compactification based on various interestingness criteria in the IDV and AGiNT systems [TPS07], [PGS06]. Introduction of definitions is a common part of state-of-the-art first-order ATPs, used to compute efficient clause normal forms [NW01]. Introduction of definitions is also an important part of unfold-definition-fold transformation in logic programming[1], the main purpose there is usually speed-up of the logic programs (reducing number of computation steps).

The work presented here tries to help understanding of formal proofs by automated finding of repeated patterns in the proofs, suggesting new definitions that capture the patterns and shorten the proofs, and help to develop a structured theory. We believe that this approach might not only help mathematicians to better understand the long automatically found proofs, but also that following the recent experiments with meta-systems for automated reasoning in large structured theories [USPV08] this approach

*Supported by the institutional grant MSM 6840770038.
†Supported by the institutional grant MSM 0021620839 and by the GAČR grant 201/08/P056.
‡Supported by the NWO project "MathWiki a Web-based Collaborative Authoring Environment for Formal Proofs".
[1]It was firstly well defined in [TS84] then extended in [PP95] and also automated in [Vv07]

could provide another way to attack hard problems automatically by enriching the theory first with new concepts, and smart heuristic abstracting away ("forgetting about") some of the concepts' properties irrelevant for the particular proof. The last mentioned is probably not the only machine-oriented application of proof compactification: compact proofs are likely to be more easy to verify, and also to combine and transform automatically in various ways.

The structure and content of this paper is as follows. Section 2 formally describes the approach used for proof compression by invention of new definitions. In Section 3 an efficient heuristic algorithm for finding best definitions based on substitution trees is suggested and its first implementation is described. In Section 4 the implementation is evaluated on ca. 8000 proofs from the TPTP library, and on several algebraic proofs. In Section 5 several examples demonstrating the work of the algorithm are shown and discussed. Section 6 discusses the possible extensions, improvements, testing, and uses of this approach, and concludes.

2 Problem statement

As mentioned above, the problem of proof improvement and refactoring is quite wide, and it can be attacked by different methods, and by employing different criteria.

The motivation for the approach taken here is that given the original proof, it can contain a large number of “similar” complex terms (terms with a large weight). Mathematicians would typically quickly simplify such proofs by introducing suitable new concepts and notation. While it is nontrivial to tell what exactly makes a new definition mathematically plausible and worth introducing in a certain proof or theory, there is at least one well-defined and easy-to-use measure of the “plausibility” of a new definition, namely the degree in which it reduces weight of the particular proof. The problem then is to find the definitions that suitably generalize the largest number of similar terms, or more precisely, to find definitions that have the best value in terms of decreasing the overall weight of the proof after replacing terms with the newly defined symbol.

The precise definition of the problem is as follows.

2.1 Problem of proof compression by new definitions

2.1.1 Proof:

A formal mathematical *proof* is understood generally, as a sequence (list, or DAG, tree, etc.) of formulae (or sequents, or just arbitrary Prolog terms) connected by inference rules. The inference rules are not relevant for the initial approach, only the formulae matter. For the purpose of this work, it suffices to treat proofs as (a set of) arbitrary Prolog terms over some initial signature of symbols (e.g., predicate and function symbols used in the proof, and logical connectives). Particular instance of this approach are the first-order proofs written in the TPTP language[2], which are just sequences of first-order TPTP formulae and clauses (written as Prolog terms) annotated with their inference information. The input data for our algorithm are then just the formulae and clauses (set of Prolog terms), without the inference data.[3]

[2]`http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`

[3]Note that the bound variables in TPTP first-order formulae are represented by Prolog variables in the TPTP syntax, however, these variables are not really "free" in the Prolog (and also first-order) sense. A proper treatment for our algorithm would be to e.g., rename such bound variables to de Bruijn indices, however the first version of our algorithm does not do this. This treatment is suboptimal, in the sense that a definition with a redundant variable can be introduced, however this has no impact on the correctness of the algorithm.

2.1.2 Weight:

A *weight assignment* w is a function from the proof signature to integers, together with an integer value for variables. The *weight* of a proof signature symbol or variable is equal to the value of the weight assignment function on it. The weight of a term, formula, or proof is a sum of the weights of the symbols and variables contained in it. The weights of symbols, terms, formulae, and proofs under a particular weight assignment w are denoted $w(s)$, $w(t)$, $w(F)$, $w(P)$. Unless specified otherwise, the simple "symbol-counting" weight assignment giving value 1 to all symbols and variables will be used as default in what follows.

2.1.3 New definition:

Given a proof (Prolog term) P, a *new definition D wrt P* is a binary Prolog clause D of the form

$$s(X_1,...,X_n)\text{:- } T(X_1,...,X_n) \tag{D}$$

where s is a symbol not appearing in P, $T(X_1,...,X_n)$ is a Prolog term over the signature of P, and $0 \leq n$. Note that this approach does not allow recursive definitions, and does not allow new variables in the body of the definition (otherwise the *most compressing definition* problem below becomes Turing-complete). Unless specified otherwise, we will also require that with a given weight assignment w, the definition D satisfies the strict monotonicity condition

$$w(s(X_1,...,X_n)) < w(T(X_1,...,X_n))$$

2.1.4 Definition application at a position:

When S is a term matching the body of the definition D (ie., there is a substitution σ such that $T(X_1,...,X_n)\sigma = S$), then $D(S)$ will denote the replacing of S by the appropriately instantiated head of the definition (ie., $s(X_1,...X_n)\sigma$, where σ is as above). Similarly, $D(P|_p)$ will denote the (unique) replacement of the subterm at position p in a term P.

2.1.5 Exhaustive definition application on the whole term:

Now consider the following definition

$$s(X) \text{ :- } f(f(X)) \tag{D_1}$$

and the term

$$f(f(f(a))) \tag{P_1}$$

Then D_1 can be applied either at the topmost position, yielding $s(f(a))$, or at the first subterm, yielding $f(s(a))$. However simultaneous application at both positions is not possible. In both cases, the default weight of the original term decreased from 4 to 3. Then consider the term

$$f(f(f(f(a)))) \tag{P_2}$$

The first application of D_1 can now be done at three different positions, yielding $s(f(f(a)))$, $f(s(f(a)))$, and $f(f(s(a)))$. For the first and third result, D_1 can be applied again, yielding $s(s(a))$ with weight 3 in both cases, while the second result with weight 4 cannot be further reduced using D_1. Hence the order of application of the definitions matters. The notation $D^*(P)$ will therefore denote any of the (possibly many and different) exhaustive applications of definition D to term P, i.e., $D^*(P)$ is a term where D can no longer be applied at any position. $D^*_{min_w}(P)$ (or just $D^*_{min}(P)$ when the weight assignment is fixed) will denote those exhaustive applications (again, generally many) such that the weight of the resulting term is minimal. Note (on terms P_1 and P_2 and definition D_1) that $D^*(P)$ and $D^*_{min}(P)$ are not unique, and can be obtained by different application paths, however in what follows we will be interested mostly only in the minimal weight and irreducibility by D.

2.1.6 The proof compression problems:

There are several well-defined problems in this setting. The *most compressing definition* problem is, for a given proof P to find the new definition D wrt to P that compresses the proof most, i.e., $w(D) + w(D^*_{min}(P))$ is minimal across all possible definitions D. Since $D^*_{min}(P)$ is non-deterministic, in practice this problem also includes finding the particular sequence of applications of D to P that result in a particular $D^*_{min}(P)$.

The *greatest proof compression* problem is, to find a set of definitions $D_1, \dots, D_n$ and a sequence of their combined applications $D^*_{1..n}(P)$ such that $w(D_1) + \dots + w(D_n) + w(D^*_{1..n}(P))$ is minimal wrt to a given proof P and weight assignment w. Let us again denote by $D^*_{1..n_{min}}(P)$ the sequences of definition applications for which this final measure is minimal. In this setting, the definitions can have in their bodies the symbols newly introduced by earlier definitions, however mutual recursivity is not possible, because the definitions applied first cannot refer to the symbols introduced later.

There are two ("greedy") ways to make the general greatest proof compression problem simpler and efficiently implementable. The first simplification consists in restricting the search space to only those sequences of definition applications where each new definition is applied exhaustively, before another new definition is considered. So the sequence of definition applications is then determined by an initial linear ordering of the set of definitions. This restriction can obviously result in worse proof compression than is possible in the general case that allows mixed application of the definitions.

The second simplification applies greediness once more, restricting the initial linear ordering of the set of definition to be done according to the compression power of the definitions. This means that first the most compressing definition D_1 is exhaustively applied to the proof, yielding a new proof $D^*_{1_{min}}(P)$ together with the added clause D_1. Let us denote this new proof P_1. Then again, the most compressing definition D_2 is found for P_1 (containing also D_1), and added and applied exhaustively, yielding proof P_2. This greedy process generates (provided all weights are positive and definitions monotone wrt w) a finite sequence of definitions and proofs. The final proof P_n can then no longer be compressed by introducing any new definition. This greedy algorithm, based on efficiently approximated algorithm for finding the most compressing definition is the basis for our implementation and experiments.

2.1.7 Is compressed proof really a proof?

One could argue that, after performing compression on a proof, the result is not a proof anymore. Consider, for example, the following fragment of a resolution proof:

$$\dots,\ a \mid b,\ \neg a,\ b,\ \dots$$

Using the definition $d = a \mid b$, we obtain

$$\dots,\ d,\ \neg a,\ b,\ \dots$$

Strictly speaking, this is not a resolution proof anymore, the inference is broken. The way we understand a compressed sequence as a proof is, using "macro-inferences", which means inference rules that, first, expand all occurences of definitions, then perform the original inference, and finally fold the result using the definitions.This is a common phenomenon when dealing with formal proofs and their presentation in e.g. formal proof assistants: Some knowledge (typically the rewriting and the definitional knowledge) is applied implicitly, without explicit reference to it and its explicit application.

2.2 Motivating example

Let's work out an example of the most compressing definition in a very simple setting: let the input consist of a single term

$$f(f(\ldots(f(a)))),$$

or shortly $f^n(a)$, for a single unary symbol f, constant a and some n. The weight of the term is $n+1$. Any compressing definition D has to be

$$d(X) = f^m(X)$$

for some m, and the shortest compression $D^*_{min}(f^n(a))$ is, up to the order of function symbols,

$$d^{n \bmod m} f^{n \operatorname{div} m}(a).$$

The weight of the definition is $m+4$, the weight of the resulting term is $n \operatorname{div} m + n \bmod m + 1$. Hence, finding the most compressing definition is equivalent to finding m minimizing the expression

$$m + n \operatorname{div} m + n \bmod m.$$

This problem has obviously polynomial complexity with respect to the input size, but it suggests that arithmetic can be involved.

3 Implementation

3.1 The most compressing definition and the least general generalization

First it is necessary to describe all the possible candidates for a new definition, and count their number (note that we are searching only for the bodies of the definitional clauses, because the heads are formed by a new symbol and a list of free variables occurring in the body). Searching for the most compressing definition for a set of terms M (representing a given proof) corresponds to searching for some least general generalization (*lgg* - see [Plo69] for exact definition) over a subset of all subterms of M. Not all compressing definitions are *lgg*'s, and not even the most compressing one has to be an *lgg*, however, the latter case is very rare. Our heuristics for compressing proofs will be based on searching for the most compressing *lgg*.

Let us look at an example which shows limits of our approach. Let M consist of a single term

$$f(g(X,\ldots,X),\ldots,g(X,\ldots,X)),$$

where f is an n-ary symbol, g is an m-ary symbol, and the total weight is $(m+1)\cdot n+1$. The *lgg* set consists of the following terms:

$$\{f(g(X,\ldots,X),\ldots,g(X,\ldots,X)), g(X,\ldots,X), X\}$$

Now, there are two reasonable candidates for the most compressing definition:

- $d(X) = f(X,\ldots,X)$. Then $D^*_{min}(M) = d(g(X,\ldots,X))$ and the total weight is $m+n+6$.

- $d(X) = g(X,\dots,X)$. Then $D^*_{min}(M) = f(d(X),\dots,d(X))$ and the total weight is $m+2n+5$.

So, if $n = 1$, both definitions are the most compressing, while for $n > 1$ the first definition wins. However, *lgg* always gives the second one.

3.2 Finding the most compressing definition using substitution trees

In this subsection, we describe our heuristics for the problem of finding the most compressing definition for a set of terms M. Our approach is based on a data structure called *substitution tree* (see [Gra96]), which has several useful properties:

1. Substitution trees are standard way to effectively save all subterms from M.
2. All nodes of the tree then always represent the use of *lgg* on a subset of all subterms of M. Moreover, there is always a tree containing a node that represents the body of the most compressing definition.
3. From the tree it is possible to quickly compute the upper estimate of the efficiency of the proof compression in the case of using a particular node as the body of the definition.

Now we will describe Algorithm 1. The input is a set of terms that correspond to some proof. The output is a term, an approximation of the most compressing definition. When such a definition does not exist, the algorithm returns "fail". At line 7 the variable U is used to denote all subterms from the input. Then a substitution tree T is created from U. T additionally remembers in its leaves the frequency of the occurrences of terms from U. At line 9, procedure propagate_freq_into_tree is called with T, described in Algorithm 2. This procedure recursively adds to each node of T the frequency corresponding to the sum of its children. From this information it is possible to compute the upper estimate for the number of application of the definitions that correspond to the node of T

The function at line 10 described at Algorithm 3 counts recursively the gain from the variable in all nodes of T that appear in the substitutions at the left-hand side. In the leaves the gain is computed from each variable at the left-hand side of the substitution as the frequency of the leaf times the weight of the term on the right hand side of each substitution where the weight of the term is computed using the function w. In the nodes that are not leaves the gain from each variable on the left-hand side is computed as the weight of the term on the right-hand side, where the weight of the term is determined using the function h (see line 11 in Algorithm 3). The gains for function h are computed by merging the gains of the node's children by summing the values at the same variables (lines 5, 6 in Algorithm 3). The gains obtained in this way will be used for fast computing of the estimate of the efficiency of the searched-for definition (lines 11 to 20 in Algorithm 1).

Now we will describe the upper estimate of the efficiency of the definition given by node N in the tree T. First we create the definition D corresponding to the node. This is done by composing all substitutions from the root to N. The resulting substitution will define the body of the definition, and all its substitutional variables, and variables that appeared in the terms inserted into the tree (these variables have to be distinguished, see [Gra96]) will be defined as the arguments in the head of the definition

The upper estimate of the definition's efficiency - the definition's gain (i.e. the upper estimate of w(proof before application of the definition) - w(proof after application of the definition) is described at lines 15 to 19.

Upper estimates are computed, because an exact computing of the definition's gain is quite inefficient (we have to go through the whole proof, and apply the definition). On the other hand, the computation of the upper estimate of all nodes from T in the way described above has the same complexity as just building the substitution tree T.

Algorithm 1 the most compressing definition

```
1.  function most_compressing_definition (proof : set of terms) : term;
    % this function returns the most compressing definition as term of the form
    % $def(x_1,...,x_n)$:- $T(x_1,...,x_n)$ of input proof where $x_1,...x_n$ are variables,
    % T is some term with at least one occurrence of every variable $x_1,...x_n$ and
    % $def$ is a new function symbol. If there is no compressing definition of proof
    % then the function returns fail.
2.  var
3.      $U$ : multiset of terms;
4.      $T$ : substitution tree;
5.      $L$ : list of tuples of the form:
              ⟨gain : integer, tag: (upper_bound, exact), definition : term⟩;
6.  {
7.      $U$ := union of all subterms of every element of proof;
8.      $T$ := construct a substitution tree from $U$ with frequencies of all terms from $U$ in leaves;
9.      propagate_freqs_into_tree($T$);
10.     (root $T$).substitution_gain := propagate_gains_of_substitutions_into_tree($T$);
11.     $L$ := empty;
12.     for each node $N$ of tree $T$ do {
13.         L := L + $\langle G, \text{upper\_bound}, D\rangle$ where                % concatenates tuple to list
14.             $D$ := create_definition_form_node($N$),
15.             $G$ := ($N$.freq$-1$)$*k(D)$ + ($j(D,N)$) - $p(D)$ where
16.                 $k(d:\text{-}\,b) := w(b) - w(d)$,                              % definition gain
17.                 $j(d:\text{-}\,b, v) := h'(b) - h'(d)$ where                    % definition gain of subst.
18.                     $h'(x) \begin{cases} n & \text{if } \langle x,n\rangle \in v.\text{substitution\_gain} \\ w(x) & \text{otherwise} \end{cases}$,
19.                 $p(d:\text{-}\,b) := w(d:\text{-}\,b)+1$,                       % penalization of def. declaration
20.     }
21.     sort $L$ with decreasing order by gain, tag where tag exact > tag upper_bound;
22.     while ($L$[1].tag = upper_bound) and ($L$[1].gain>0) do {
23.         $L$[1].gain := calculate the exact $L$[1].definition gain as: $w$(proof) - $w$(greedy application of
    $L$[1].definition on proof) - $p$(L[1].definition) where
24.             $p(d:\text{-}\,b) := w(d:\text{-}\,b)$+1;                           % penalization of def. declaration
25.         $L$[1].tag := exact;                                  % changes tag of the first element of list L
26.         $L$ := merge L[1] with L[2..];
    % merges the first element of list with its tail by the same rules as at 21.
27.     }
28.     if  $L$[1].gain > 0  then  return $L$[1].definition  else  return fail;
29. }
```

Algorithm 2 propagate frequencies into tree

```
1. procedure propagate_freqs_into_tree ($T$ : substitution_tree) ;
2. {
3.     if $T$ is leaf then exit;                    % Frequency of leaf is already calculated.
4.     (root $T$).substitution_gain := empty;
5.     for each son $S$ of root $T$ do {
6.         propagate_freqs_into_tree(subtree of $T$ where $S$ is root);   % Calculates freqs in subtree $S$.
7.         (root $T$).freq := (root $T$).freq + $S$.freq;
8.     }
9. }
```

The resulting estimate is inserted into the list L as a tuple $\langle$upper estimate, tag: upper_bound, definition $D\rangle$ (see line 13). After inserting all the upper estimates of all the nodes of T, the list L is sorted decreasingly by the size of the upper estimate. If the upper estimates are equal, the tuple with tag "exact" is preferred to the tuple with tag "upper_bound".

Now we always test if the first member of the list L is already an exact efficiency value denoted with the tag "exact". If so, the value D in this member is the searched-for most compressing definition. This definition is the best among all the nodes of the tree T, but it does not have to be the best definition absolutely, because we are selecting only from the nodes of the tree T. If the value is not exact, we compute the exact value of D for the proof (by replacing). The result is saved as the first element of the list, and is tagged "exact". Then we sort the list, and repeat, see lines 22 to 27.

If the resulting gain is more than 0, it means that the applied definition shortens the proof and this definition is returned at the output. If not, we return "fail", because no compressing definition appears among the nodes of T.

Algorithm 4 describes the greedy approach (see Subsection 2.1.6) for finding an approximation of the greatest proof compression.

4 Testing

The initial implementation described above has been tested on the whole TPTP library [Sut09], and on two families of proofs coming from recent research in algebra. In both cases the simplest symbol-counting weight function was used for measuring the proof improvement.

4.1 Testing on the TPTP Library

The TPTP library contains a large number of ATP problems from various areas of ATP use, which makes it suitable for large-scale evaluation of the proof compression algorithm. For the testing described here, TPTP version 4.0.1 was used, containing 16512 problems. All available TPTP proofs found by the EP version 1.1 theorem prover [Sch02] were obtained from the TPTP developer Geoff Sutcliffe. This is a testing set of 7807 ATP proofs in the TPTP syntax, which is a subset of the Prolog syntax. These proofs were postprocessed by a simple Perl script into a list of formulae (i.e., forgetting about the inference

Algorithm 3 propagate gains of substitutions into tree

```
 1. function propagate_gains_of_substitutions_into_tree ($T$ : substitution_tree)
             : set of couples of the form: ⟨var : subst. variable, gain : integer⟩ ;
 2. var $R$ : set of couples of the form: ⟨var : substitution variable, gain : integer⟩ ;
 3. {
 4.     (root $T$).substitution_gain := empty;        % there are no subst. variables in leaves.
 5.     for each son $S$ of root $T$ do {
 6.         (root $T$).substitution_gain := merge (root $T$).substitution_gain with propagate_gains_of_substitutions_into_tree(subtree of $T$ where $S$ is root) so that, couples with the same variable is merged into one couple and its gain is a sum of gains of all original couples with the same variable;
 7.     }
 8.     $R$ := empty;
 9.     for each substitution $\theta$=$T$ of substitution set in root $T$ do {
10.         $R$ := $R\cup\langle\theta,h(T)\rangle$ where
11.             $h(x)\begin{cases} n & \text{if } \langle x,n\rangle \in \text{(root } T\text{).substitution\_gain} \\ w(x) & \text{otherwise} \end{cases}$;
12.     }
13.     if $T$ is leaf then
14.         for each couple ⟨x,n⟩ of $R$ do {
15.             $R$ := ($R$ \ ⟨x,n⟩)∪⟨x, n*(root $T$).freq⟩;
16.         }
17.     return $R$;
18. }
```

structure). This again can be considered to be just a list of Prolog terms, and hence it is already an input to the proof compression algorithm explained above.

The testing was done on an eight-core 64bit Intel Xeon E5520 2.27 GHz Linux machine with 8GB of RAM. SWI Prolog was used to run the proof compression algorithm. SWI Prolog has some internal memory limits that do not allow it to get past 2GB boundary, so for very large proofs the implementation can now fail for lack of memory. Because the implementation can also take quite a long time for large proofs (the longest example we are aware of was about one hour), we have given each of the TPTP proofs a time limit of 60 seconds to be able to finish the large-scale testing in reasonable time. 4890 of the 7807 proofs (63%) were completely compressed within the time limit, i.e., the algorithm has successfully finished in 60 seconds. For the remaining 2917 proofs the algorithm typically has found the initial most compressing definitions, but has not converged to the point where no more compressing definitions exist. The final compression ratios for the 4890 successful runs can be viewed online at our webpage[4], and all the TPTP proofs together with the algorithm inputs and outputs can also be viewed

[4]`http://mws.cs.ru.nl/~urban/compression/00tstp_final_ratios`

Algorithm 4 the greatest proof compressing

```
1.  function greatest_proof_compressing (proof : set of terms) : set of terms;
2.  var
3.      $R$ : set of terms;
4.      $T$ : definition;
5.  {
6.      $R$ := proof;
7.      $T$ := most_compressing_definition($R$);
8.      while $T \neq$ fail do {
9.          $R$ := $T \cup$(application of definition $T$ on $R$);
10.         $T$ := most_compressing_definition($R$);
11.     }
12.     return $R$;
13. }
```

TPTP problems	proved by EP	compressed in 60s	timeout in 60s
16512	7807	4890	2917
greatest comp. ratio	least comp. ratio	median ratio	comp. below 50%
0.1844	0.9969	0.8100	135

Table 1: Results of testing the proof compression algorithm on the TPTP library

there[5]. The interesting data extracted from the testing are summarized in Table 1, and Figure 1 shows the graph of the compression performance on the 4890 finished proofs.

4.2 Testing on algebraic problems

One of the aspects of the present work is, to address the problem of human understanding of machine generated proofs. For this reason, we tested our implementation on two families of proofs, coming from different areas of current research in algebra.

4.2.1 Loops with abelian inner mappings

We investigated a proof, obtained by Waldmeister, that every uniquely 2-divisible loop with abelian inner mapping group of exponent 2, is commutative and associative [PS08][6]. In both cases, the very first definition the implementation found, was the right inverse operation (that is, the term $1/x$), and the left inverse followed soon. Other interesting definitions were shortcuts for various compositions of the inner mappings. Both proofs had final ratio about 0.75.

[5]http://mws.cs.ru.nl/~urban/compression/Solutions_tstp/

[6]http://mws.cs.ru.nl/~urban/compression/aim_2div/

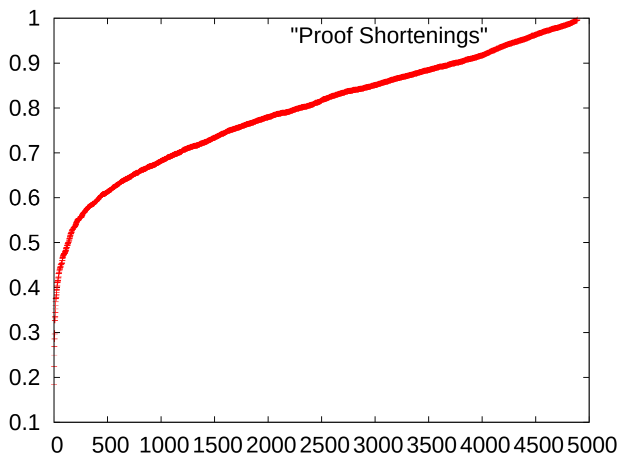

Figure 1: Proof compression ratios on the TPTP library, sorted from the best to the worst

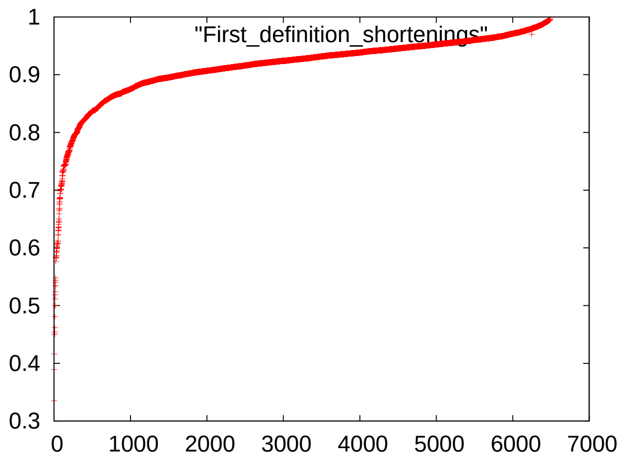

Figure 2: Proof shortening ratios by the most compressing definition on the TPTP library, sorted from the best to the worst

4.2.2 Symmetric-by-medial groupoids

We investigated three related proofs, obtained in [Sta08] with Prover9[7]. The importance of the term $xy \cdot zu$ in the theory of distributive groupoids was recognized immediately in each case. In the latter two cases, it cuts the proof weight by more than 10%. Sadly, other definitions found by the implementation seem to have little mathematical meaning. The final ratios were 0.65, 0.72, and 0.75, respectively.

4.2.3 Algebraic problems in TPTP

Many algebraic problems were recently submitted to TPTP [PS08], for instance, problems in the interval GRP654 to GRP763. Our notes in Section 5.2 are also based on the inspection of the results on these problems.

[7] `http://mws.cs.ru.nl/~urban/compression/symbymed/`

5 Examples and discussion

5.1 A good and a bad example

To get a taste of the results, we shall look closer at one of the most successful, and one of the most unsuccessful compressions produced by our implementation on the TPTP problems.

The champion is SWV158, with final compression ratio 0.1844 (from 2277 to 420)[8]. The first definition is of enormous weight, 86, and its application saves almost half of the proof weight. This is an equality, with a variable on one side, and a very nested term on the other side, with many constant leaves and just one free variable with 6 occurrences. Three more heavy definitions of a similar kind, and 10 lighter ones, finish the compression.

To the other extreme, let's mention GRP754, with final compression ratio 0.9710 (from 726 to 705)[9]. There is a single compressing definition, setting `def1(A, B, C)= (A=mult(B, C))`, which is applied on roughly two thirds of the proof lines.

Generally speaking, heavy definitions are rare. Most definitions save just very few symbols, but are applied many times in the proof.

5.2 Understanding machine generated proofs

Our experiments show that introducing a new definition that formally reduces weight of the proof, rarely gives a notion interesting from mathematician's point of view. Interesting exceptions exist: the implementation discovered notions like left and right inverse, and in some cases isolated important concepts that occur frequently in the proof.

Yet, reading the result of the overall greedy algorithm, it is a mess. The problem seems to have several layers. First of all, only few definitions have a good mathematical meaning. It is desirable, to introduce other measures, to judge which definitions are "good" and which are "bad", perhaps in the spirit of AGInT [PGS06]. The most compressing criterion is a reasonable heuristic, but far from perfect.

Another aspect is that, for human readers, learning new definitions is costly. In fact, looking at the examples, we realized that many definitions save just one character, even the top ones (their choice by the algorithm comes from the fact that they can be used many times). Perhaps we shall add a penalty to each new definition, based on how difficult is it to grasp it, relatively to how useful it is. Too short definitions, or those that are used only few times, shall be discarded.

One particular example of "bad" definition is the following. For the sake of simplicity, assume the signature consists of a single binary function symbol f. Then, (almost) any proof can be simplified introducing the predicate $P(x,y,z)$, defined by $f(x,y)=z$, saving one symbol per (almost every) line. Further in this direction, the formulae in the proof are actually very likely to be in the form $f(\_,\_) = f(\_,\_)$, and the corresponding 4-ary predicate symbol shortens the proof by 5/7. Indeed, such definitions don't bring any better understanding. The weight function, or the "beauty criterion", shall avoid this sort of definitions.

6 Future work and conclusions

The presented system provides a useful means for experimenting with introducing new definitions based on the frequency and weight of subterms in a proof. The general problem of greatest proof compression seems to be quite hard, however our heuristic greedy implementation seems to perform reasonably well

[8]`http://mws.cs.ru.nl/˜urban/compression/Solutionststp/SWV/SWV158+1/`

[9]`http://mws.cs.ru.nl/˜urban/compression/Solutionststp/GRP/GRP754-1/`

already in its first version. It seems to be an interesting open problem, to determine the complexity class of finding the greatest proof compression.

The initial evaluation using the most straightforward weight assignment has allowed us to immediately see some deficiencies in overly greedy introduction of new definitions. The system is sometimes capable of identifying mathematically interesting concepts that significantly compress the proofs, however many times the introduced definitions seem to be of little mathematical value, and only complicate the proof understanding. As already mentioned above, this will likely lead to further research about the proper offset between the benefits of the proof compression, and the benefits of not having to deal with too many similar concepts in one's head. It is obvious that shorter proofs don't always have to be "nicer" (whatever it means), but it is obviously also good to have tools that can produce the best result according to a precisely defined criterion.

The advantage of our system is that a lot of experimenting can be done using the weight assignments. For example, we could try:

- to weight equality symbol with zero (this is sufficient to avoid definitions of the form $f(x,y) = z$),
- to add learning penalties, for instance, by setting the weight of a new symbol by the maximal weight of symbols in its defining term, plus one,
- try to learn weight assignment patterns by data-mining techniques on a large body of available structured mathematics, e.g., the formal Mizar library.

The last option even suggests some more interesting experiments in the context of a large formal body of human-written mathematics. For example, it is feasible (using the MPTP system) to expand the whole Mizar library (or a suitable part of it) into a basic set-theoretical notation, i.e., using just the membership and equality predicates, and eliminating all definitions introduced by humans. This will likely result in a very large, but manageable (e.g. with complete term sharing in the implementation) blow-up of the library. Then the system can be used to search for interesting definitions automatically, and the results can be compared with the definitions introduced by human authors.

Another potential use that we are very interested in, is the use of the subsystem as a "concept developing" component in larger meta-systems (like MaLARea [USPV08]) for theorem proving in structured and large theories. The experience with ATP in large theories so far shows that blind recursive inclusion of all available definitions and all the theorems known about them significantly decreases the ATP performance in the large theories. Introducing new concepts, and only selecting some of their important properties (like commutativity) is also a very common feature of abstract mathematical developments done by human mathematicians. Both the human evidence, and the evidence from doing ATP in large theories therefore points to the importance of including good concept creation into the overall process of mathematical theorem proving and theory development.

7 Acknowledgements

We would like to thank the anonymous LPAR referees for thorough reading, and a number of useful comments and suggestions to the first version of the paper. Thanks to Geoff Sutcliffe for providing the E proofs of the TPTP problems.

References

[Dah98] Ingo Dahn. Robbins algebras are boolean: A revision of mccune's computer-generated solution of robbins problem. *Journal of Algebra*, 208(2):526–32, 1998.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[Gra96] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer, 1996.

[Gra01] Adam Grabowski. Robbins algebras vs. boolean algebras. *Formalized Mathematics*, 9(4):681–690, 2001.

[Hua96] Xiaorong Huang. Translating machine-generated resolution proofs into nd-proofs at the assertion level. In Norman Y. Foo and Randy Goebel, editors, *PRICAI*, volume 1114 of *Lecture Notes in Computer Science*, pages 399–410. Springer, 1996.

[McC97] William McCune. Solution of the Robbins problem. *J. Autom. Reasoning*, 19(3):263–276, 1997.

[MP08] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[NW01] A. Nonnengart and C. Weidenbach. *Handbook of Automated Reasoning*, volume I, chapter Computing small clause normal forms., pages 335–367. Elsevier and MIT Press, 2001.

[PGS06] Yury Puzis, Yi Gao, and Geoff Sutcliffe. Automated generation of interesting theorems. In *FLAIRS Conference*, pages 49–54, 2006.

[Plo69] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1969.

[PP95] Maurizio Proietti and Alberto Pettorossi. Unfolding–definition–folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1 May 1995.

[PS08] J. D. Phillips and David Stanovský. Automated theorem proving in loop theory. In Geoff Sutcliffe, Simon Colton, and Stephan Schulz, editors, *ESARM: Empirically Successful Automated Reasoning in Mathematics*, volume 378 of *CEUR Workshop Proceedings*, pages 54–54. CEUR, 2008.

[Sch02] S. Schulz. E – a brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[Sta08] David Stanovský. Distributive groupoids are symmetric-by-medial: An elementary proof. *Commentationes Mathematicae Universitatis Carolinae*, 49(4):541–546, 2008.

[Sut09] Geoff Sutcliffe. The tptp problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[TPS07] Steven Trac, Yury Puzis, and Geoff Sutcliffe. An interactive derivation viewer. In *UITP 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 109–123. Elsevier, 2007.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In Sten-Åke Tärnlund, editor, *Proceedings of The Second International Conference on Logic Programming*, pages 127–139, 1984.

[Urb08] Josef Urban. Automated reasoning for mizar: Artificial intelligence through knowledge exchange. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *LPAR Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. Malarea sg1- machine learner for automated reasoning with semantic guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2008.

[Ver01] Robert Veroff. Solving open questions and other challenge problems using proof sketches. *J. Autom. Reasoning*, 27(2):157–174, 2001.

[Vv07] Jiří Vyskočil and Petr Štěpánek. Improving efficiency of prolog programs by fully automated unfold/-fold transformation. In *MICAI*, pages 305–315, 2007.