Compression of First-Order Resolution Proofs by Partial Regularization

Jan Gorzny[1] * and Bruno Woltzenlogel Paleo[2] **

[1] jgorzny@uvic.ca, University of Victoria, Canada
[2] bruno@logic.at, Vienna University of Technology, Austria

Abstract. This paper describes a generalization of the recently developed `RecyclePivotsWithIntersection` algorithm applicable to first-order logic proofs generated by automated theorem provers. `RecyclePivotsWithIntersection` removes inferences with a node η when its pivot literal occurs as the pivot of another inference located below in the path from η to the root of the proof. The algorithm is then combined with `GreedyLinearFirstOrderLowerUnits` in order to compress first-order proofs further. A preliminary empirical evaluation of the combination of these compression algorithms is presented.

1 Introduction

Most of the effort in automated reasoning so far has been dedicated to the design and implementation of proof systems and efficient theorem proving procedures. As a result, saturation-based first-order automated theorem provers have achieved a high degree of maturity, with resolution and superposition being among the most common underlying proof calculi. Proof production is an essential feature of modern state-of-the-art provers and proofs are crucial for applications where the user requires certification of the answer provided by the prover. Nevertheless, efficient proof production is non-trivial, and it is to be expected that the best, most efficient, provers do not necessarily generate the best, least redundant, proofs. Therefore, it is a timely moment to develop methods that post-process and simplify proofs. While the foundational problem of simplicity of proofs can be traced back at least to Hilbert's 24th Problem, the maturity of automated deduction has made it particularly relevant today.

For proofs generated by SAT- and SMT-solvers, which use propositional resolution as the basis for the DPLL and CDCL decision procedures, there is now a wide variety of proof compression techniques. Algebraic properties of the resolution operation that might be useful for compression were investigated in [6]. Compression algorithms based on rearranging and sharing chains of resolution inferences have been developed in [1] and [11]. Cotton [5] proposed an algorithm that compresses a refutation by repeteadly splitting it into a proof of a heuristically chosen literal ℓ and a proof of $\overline{\ell}$, and then resolving them to form a

* Supported by the Google Summer of Code 2014 program.
** Supported by the Austrian Science Fund, project P24300.

new refutation. The `Reduce&Reconstruct` algorithm [10] searches for locally redundant subproofs that can be rewritten into subproofs of stronger clauses and with fewer resolution steps. A linear time proof compression algorithm based on partial regularization was proposed in [2] and improved in [7]. Furthermore, [7] also described a new linear time algorithm called `LowerUnits`, which delays resolution with unit clauses.

In contrast, for first-order theorem provers, there has been up to now (to the best of our knowledge) no attempt to design and implement an algorithm capable of taking a first-order resolution DAG-proof and efficiently simplifying it, outputting a possibly shorter pure first-order resolution DAG-proof. There are algorithms aimed at simplifying first-order sequent calculus tree-like proofs, based on cut-introduction [9, 8], and while in principle resolution DAG-proofs can be translated to sequent-calculus tree-like proofs (and then back), such translations lead to undesirable efficiency overheads. There is also an algorithm [13] that looks for terms that occur often in any TSTP [12] proof (including first-order resolution DAG-proofs) and introduces abbreviations for these terms. However, as the definitions of the abbreviations are not part of the output proof, it cannot be checked by a pure first-order resolution proof checker.

In this paper, we initiate the process of lifting propositional proof compression techniques to the first-order case, starting with the simplest known algorithm: `LowerUnits` (described in Section **??**). As shown in Section 3, even for this simple algorithm, the fact that first-order resolution makes use of unification leads to many challenges that simply do not exist in the propositional case. In Section **??** we describe an algorithm that overcomes these challenges. As this algorithm has quadratic run-time complexity with respect to the input proof length, in Section **??** we describe a variation of this algorithm, which has linear run-time complexity and is easier to implement, at the cost of compressing less. In Section 5 we present experimental results obtained by applying this algorithm on hundreds of proofs generated with the SPASS theorem prover. The next section introduces the first-order resolution calculus using notations that are more convenient for describing proof transformation operations.

2 The Resolution Calculus

We assume that there are infinitely many variable symbols (e.g. X, Y, Z, X_1, X_2, ...), constant symbols (e.g. a, b, c, a_1, a_2, ...), function symbols of every arity (e.g f, g, f_1, f_2, ...) and predicate symbols of every arity (e.g. p, q, p_1, p_2,...). A *term* is any variable, constant or the application of an n-ary function symbol to n terms. An *atomic formula* (*atom*) is the application of an n-ary predicate symbol to n terms. A *literal* is an atom or the negation of an atom. The *complement* of a literal ℓ is denoted $\overline{\ell}$ (i.e. for any atom p, $\overline{p} = \neg p$ and $\overline{\neg p} = p$). The set of all literals is denoted $\mathcal{L}$. A *clause* is a multiset of literals. $\bot$ denotes the *empty clause*. A *unit clause* is a clause with a single literal. Sequent notation is used for clauses (i.e. $p_1, \ldots, p_n \vdash q_1, \ldots, q_m$ denotes the clause $\{\neg p_1, \ldots, \neg p_n, q_1, \ldots, q_m\}$). $\mathrm{FV}(t)$ (resp. $\mathrm{FV}(\ell)$, $\mathrm{FV}(\Gamma)$) denotes the set of variables in the term t (resp. in the

literal ℓ and in the clause Γ). A *substitution* $\{X_1\backslash t_1, X_2\backslash t_2, \ldots\}$ is a mapping from variables $\{X_1, X_2, \ldots\}$ to, respectively, terms $\{t_1, t_2, \ldots\}$. The application of a substitution σ to a term t, a literal ℓ or a clause Γ results in, respectively, the term $t\sigma$, the literal $\ell\sigma$ or the clause $\Gamma\sigma$, obtained from t, ℓ and Γ by replacing all occurrences of the variables in σ by the corresponding terms in σ. The set of all substitutions is denoted $\mathcal{S}$. A *unifier* of a set of literals is a substitution that makes all literals in the set equal. A *resolution proof* is a directed acyclic graph of clauses where the edges correspond to the inference rules of resolution and contraction (as explained in detail in Definition 1). A *resolution refutation* is a resolution proof with root $\bot$.

Definition 1 (First-Order Resolution Proof).
A directed acyclic graph $\langle V, E, \Gamma\rangle$, where V is a set of nodes and E is a set of edges labeled by literals and substitutions (i.e. $E \subset V \times 2^{\mathcal{L}} \times \mathcal{S} \times V$ and $v_1 \xrightarrow[\sigma]{\ell} v_2$ denotes an edge from node v_1 to node v_2 labeled by the literal ℓ and the substitution σ), is a proof of a clause Γ iff it is inductively constructible according to the following cases:

- ***Axiom:*** *If Γ is a clause, $\widehat{\Gamma}$ denotes some proof $\langle\{v\}, \varnothing, \Gamma\rangle$, where v is a new (axiom) node.*
- ***Resolution:*** *If ψ_L is a proof $\langle V_L, E_L, \Gamma_L\rangle$ with $\ell_L \in \Gamma_L$ and ψ_R is a proof $\langle V_R, E_R, \Gamma_R\rangle$ with $\ell_R \in \Gamma_R$, and σ_L and σ_R are substitutions such that $\ell_L\sigma_L = \overline{\ell_R}\sigma_R$ and $\mathrm{FV}((\Gamma_L \setminus \{\ell_L\})\,\sigma_L) \cap \mathrm{FV}((\Gamma_R \setminus \{\ell_R\})\,\sigma_R) = \emptyset$, then $\psi_L \odot_{\ell_L\ell_R}^{\sigma_L\sigma_R} \psi_R$ denotes a proof $\langle V, E, \Gamma\rangle$ s.t.*

$$
\begin{aligned}
V &= V_L \cup V_R \cup \{v\}\\
E &= E_L \cup E_R \cup \left\{\rho(\psi_L) \xrightarrow[\sigma_L]{\{\ell_L\}} v, \rho(\psi_R) \xrightarrow[\sigma_R]{\{\ell_R\}} v\right\}\\
\Gamma &= (\Gamma_L \setminus \{\ell_L\})\,\sigma_L \cup (\Gamma_R \setminus \{\ell_R\})\,\sigma_R
\end{aligned}
$$

  where v is a new (resolution) node and $\rho(\varphi)$ denotes the root node of φ. The resolved atom *ℓ is such that $\ell = \ell_L\sigma_L = \overline{\ell_R}\sigma_R$ or $\ell = \overline{\ell_L}\sigma_L = \ell_R\sigma_R$.*
- ***Contraction:*** *If ψ' is a proof $\langle V', E', \Gamma'\rangle$ and σ is a unifier of $\{\ell_1, \ldots \ell_n\}$ with $\{\ell_1, \ldots \ell_n\} \subseteq \Gamma'$, then $\lfloor\psi\rfloor^{\sigma}_{\{\ell_1,\ldots\ell_n\}}$ denotes a proof $\langle V, E, \Gamma\rangle$ s.t.*

$$
\begin{aligned}
V &= V' \cup \{v\}\\
E &= E' \cup \{\rho(\psi') \xrightarrow[\sigma]{\{\ell_1,\ldots\ell_n\}} v\}\\
\Gamma &= (\Gamma' \setminus \{\ell_1, \ldots \ell_n\})\sigma \cup \{\ell\}
\end{aligned}
$$

  where v is a new (contraction) node, $\ell = \ell_k\sigma$ (for any $k \in \{1, \ldots, n\}$) and $\rho(\varphi)$ denotes the root node of φ. □

The resolution and contraction (factoring) rules described above are the standard rules of the resolution calculus, except for the fact that we do not require resolution to use most general unifiers. The presentation of the resolution rule

here uses two substitutions, in order to explicitly handle the necessary renaming of variables, which is often left implicit in other presentations of resolution.

When we write $\psi_L \odot_{\ell_L \ell_R} \psi_R$, we assume that the omitted substitutions are such that the resolved atom is most general. We write $\lfloor \psi \rfloor$ for an arbitrary maximal contraction, and $\lfloor \psi \rfloor^\sigma$ for a (pseudo-)contraction that does merge no literals but merely applies the substitution σ. When the literals and substitutions are irrelevant or clear from the context, we may write simply $\psi_L \odot \psi_R$ instead of $\psi_L \odot^{\sigma_L \sigma_R}_{\ell_L \ell_R} \psi_R$. The $\odot$ operator is assumed to be left-associative. In the propositional case, we omit contractions (treating clauses as sets instead of multisets) and $\psi_L \odot^{\emptyset\emptyset}_{\ell\bar{\ell}} \psi_R$ is abbreviated by $\psi_L \odot_\ell \psi_R$.

If $\psi = \varphi_L \odot \varphi_R$ or $\psi = \lfloor \varphi \rfloor$, then φ, φ_L and φ_R are *direct subproofs* of ψ and ψ is a *child* of both φ_L and φ_R. The transitive closure of the direct subproof relation is the *subproof* relation. A subproof which has no direct subproof is an *axiom* of the proof. V_ψ, E_ψ and Γ_ψ denote, respectively, the nodes, edges and proved clause (conclusion) of ψ. If ψ is a proof ending with a resolution node, then ψ_L and ψ_R denote, respectively, the left and right premises of ψ.

3 First-Order Challenges

In this section, we describe challenges that have to be overcome in order to successfully adapt `RecyclePivotsWithIntersection` to the first-order case. The first example illustrates the need to take unification into account. The other two examples discuss complex issues that can arise when unification is take into account in a naive way.

Example 1. Consider the following proof ψ, noting that the proof is largely redundant. Naively computed, the safe literals for η_3 are $\{\vdash q(c),\ p(a, X)\}$. η_1 and η_5 and these two literals are unifiable. Further, the safe literals for η_1 includes η_5. Thus the proof can be regularized by recycling η_1.

$$\dfrac{\dfrac{\dfrac{\eta_1\colon\ \vdash p(W, X) \qquad \eta_2\colon p(W, X)\ \vdash q(c)}{\eta_3\colon\ \vdash q(c)} \qquad \eta_4\colon q(c)\ \vdash p(a, X)}{\eta_5\colon\ \vdash p(a, X)} \qquad \eta_6\colon p(Y, b)\ \vdash}{\psi\colon \bot}$$

Regularization of the proof by recycling η_1 results in deleting the edge between η_2 and η_3, which in turn replaces η_3 by η_1. Since η_1 cannot be resolved against η_4, and η_1 contained safe literals, η_5 is replaced by η_1. The result is the much shorter proof below.

$$\dfrac{\eta_1\colon\ \vdash p(W, X) \qquad \eta_2\colon p(Y, b)\ \vdash}{\psi'\colon \bot}$$

Unlike in the propositional case, where the pivots and their corresponding safe literal list are all syntactically equal, in the first-order case, this is not necessarily the case. As illustrated above, $p(W, X)$ and $p(a, X)$ are not syntactically equal. Nevertheless, they are unifiable, and the proof can be regularized.

Example 2. There are cases, as shown below, that require more careful care when attempting to regularize. Again, naively computed the safe literals for η_3 are $\{\vdash q(c),\ p(a,X)\}$, and so η_1 and η_2 appear to be candidates for regularization.

$$\dfrac{\dfrac{\dfrac{\eta_1\colon\ \vdash p(a,c) \qquad \eta_2\colon p(a,c)\ \vdash q(c)}{\eta_3\colon\ \vdash q(c)} \qquad \eta_4\colon q(c)\ \vdash p(a,X)}{\eta_5\colon\ \vdash p(a,X)} \qquad \eta_6\colon p(Y,b)\ \vdash}{\psi\colon \bot}$$

However, if we attempt to regularize the proof, the same series of actions as in Example 1 would result in the following resolution, which cannot be completed.

$$\dfrac{\eta_1\colon\ \vdash p(a,c) \qquad \eta_2\colon p(Y,b)\ \vdash}{\psi'\colon ??}$$

The observations above lead to the idea of requiring pivots to satisfy the following property before collecting them to be reused.

Definition 2. *Let η be a clause with literal ℓ' with corresponding safe literal ℓ which is resolved against literals $\ell_1, \ldots, \ell_n$ in a proof ψ. η is said to satisfy the* pre-regularization unifiability property *in ψ if $\ell_1,\ldots,\ell_n$, and $\overline{\ell'}$ are unifiable.*

One technique to ensure this property is met is to apply the unifier of a resolution to each resolvent before computing the safe literals. In the case of Example 2, this would result in η_1 having the safe literals $\{\vdash q(c),\ p(a,b)\}$, and now it is clear that the literal in η_1 is not safe.

Example 3. Satisfaction of the pre-deletion unifiability property is not enough. Deletion of the units from a proof ψ may actually change the literals that had been resolved away by the units, because fewer substitutions are applied to them. This is exemplified below:

$$\dfrac{\dfrac{\dfrac{\eta_1\colon r(Y), p(X,q(Y,b)), p(X,Y)\vdash \qquad \eta_2\colon\ \vdash p(U,V)}{\eta_3\colon r(V), p(U,q(V,b))\vdash} \qquad \eta_4\colon\ \vdash r(W)}{\eta_5\colon p(U,q(W,b))\vdash} \qquad \eta_2}{\psi\colon \bot}$$

If η_2 is collected for lowering and deleted from ψ, we obtain the proof $\psi \setminus \{\eta_2\}$:

$$\dfrac{\eta_1'\colon r(Y), p(X,q(Y,b)), p(X,Y)\vdash \qquad \eta_4'\colon\ \vdash r(W)}{\eta_5'(\psi')\colon p(X,q(W,b)), p(X,W)\vdash}$$

Note that, even though η_2 satisfies the pre-deletion unifiability property (since $p(X,q(Y,b))$ and $p(U,q(W,b))$ are unifiable), η_2 still cannot be lowered and reintroduced by a single resolution inference, because the corresponding modified post-deletion literals $p(X,q(W,b))$ and $p(X,W)$ are actually not unifiable.

The observation above leads to the following stronger property:

Definition 3. *Let η be a unit with literal ℓ_η and let $\eta_1, \ldots, \eta_n$ be subproofs that are resolved with η in a proof ψ, respectively, with resolved literals $\ell_1, \ldots, \ell_m$. η is said to satisfy the* post-deletion unifiability property *in ψ if $\ell_1^{\dagger\downarrow},\ldots,\ell_m^{\dagger\downarrow}$, and $\overline{\ell_\eta^\dagger}$ are unifiable, where $\ell^\dagger$ is the literal in $\psi \setminus \{\eta\}$ corresponding to ℓ in ψ and $\ell_k^{\dagger\downarrow}$ is the descendant of $\ell_k^\dagger$ in the root of $\psi \setminus \{\eta\}$.*

4 First-Order RecyclePivotsWithIntersection

This section presents `FirstOrderRecyclePivotsWithIntersection` (Algorithm **??**), a first order generalization of `RecyclePivotsWithIntersection`.

Our second algorithm, `RPI` (`RPI`), aims at compressing irregular proofs. It can be seen as a simple but significant modification of the (`RPI`) algorithm described in [**?**], from which it derives its name. Although in the worst case full regularization can increase the proof length exponentially [**?**], these algorithms show that many irregular proofs can have their length decreased if a careful partial regularization is performed.

These observations lead to the idea of traversing the proof in a bottom-up manner, storing for every node a set of *safe literals* that get resolved in all paths below it in the proof (or that already occurred in the root clause of the original proof). Moreover, if one of the node's resolved literals belongs to the set of safe literals, then it is possible to regularize the node by replacing it by one of its parents (cf. Algorithm 1).

The regularization of a node should replace a node by one of its parents, and more precisely by the parent whose clause contains the resolved literal that is safe. After regularization, all nodes below the regularized node may have to be fixed. However, since the regularization is done with a bottom-up traversal, and only nodes below the regularized node need to be fixed, it is again possible to postpone fixing and do it with only a single traversal afterwards. Therefore, instead of replacing the irregular node by one of its parents immediately, its other parent is replaced by `deletedNodeMarker`, as shown in Algorithm 2. Only later during fixing, the irregular node is actually replaced by its surviving parent (i.e. the parent that is not `deletedNodeMarker`).

The set of safe literals of a node η can be computed from the set of safe literals of its children (cf. Algorithm 3). In the case when η has a single child ς, the safe literals of η are simply the safe literals of ς together with the resolved literal p of ς belonging to η (p is safe for η, because whenever p is propagated down the proof through η, p gets resolved in ς). It is important to note, however, that if ς has been marked as regularized, it will eventually be replaced by η, and hence p should not be added to the safe literals of η. In this case, the safe literals

input : A proof ψ
output: A possibly less-irregular proof ψ'

1 $\psi' \leftarrow \psi$;
2 traverse ψ' bottom-up and **foreach** *node* η *in* ψ' **do**
3 **if** η *is a resolvent node* **then**
4 setSafeLiterals(η) ;
5 regularizeIfPossible(η)
6 $\psi' \leftarrow$ fix(ψ') ;
7 **return** ψ';

Algorithm 1:

```
   input  : A node η
   output: nothing (but the proof containing η may be changed)
 1 if η.rightResolvedLiteral ∈ η.safeLiterals then
 2     replace left parent of η by deletedNodeMarker ;
 3     mark η as regularized
 4 else if η.leftResolvedLiteral ∈ η.safeLiterals then
 5     replace right parent of η by deletedNodeMarker ;
 6     mark η as regularized
```

Algorithm 2: `regularizeIfPossible`

of η should be exactly the same as the safe literals of ς. When η has several children, the safe literals of η w.r.t. a child ς_i contain literals that are safe on all paths that go from η through ς_i to the root. For a literal to be safe for all paths from η to the root, it should therefore be in the intersection of the sets of safe literals w.r.t. each child.

The `RPI` and the `RPI` algorithms differ from each other mainly in the computation of the safe literals of a node that has many children. While `RPI` returns the intersection as shown in Algorithm 3, `RPI` returns the empty set (cf. Algorithm 4). Additionally, while in `RPI` the safe literals of the root node contain all the literals of the root clause, in `RPI` the root node is always assigned an empty set of literals. (Of course, this makes a difference only when the proof is not a refutation.) Note that during a traversal of the proof, the lines from 5 to 10 in Algorithm 3 are executed as many times as the number of edges in the proof. Since every node has at most two parents, the number of edges is at most twice the number of nodes. Therefore, during a traversal of a proof with n nodes, lines from 5 to 10 are executed at most $2n$ times, and the algorithm remains linear. In our prototype implementation, the sets of safe literals are instances of Scala's `mutable.HashSet` class. Being mutable, new elements can be added efficiently.

```
   input  : A node η
   output: nothing (but the node η gets a set of safe literals)
 1 if η is a root node with no children then
 2     η.safeLiterals ← η.clause
 3 else
 4     foreach η′ ∈ η.children do
 5         if η′ is marked as regularized then
 6             safeLiteralsFrom(η′) ← η′.safeLiterals ;
 7         else if η is left parent of η′ then
 8             safeLiteralsFrom(η′) ← η′.safeLiterals ∪ { η′.rightResolvedLiteral } ;
 9         else if η is right parent of η′ then
10             safeLiteralsFrom(η′) ← η′.safeLiterals ∪ { η′.leftResolvedLiteral } ;
11     η.safeLiterals ← ⋂_{η′∈η.children} safeLiteralsFrom(η′)
```

Algorithm 3: `setSafeLiterals`

```
   input  : A node η
   output: nothing (but the node η gets a set of safe literals)
 1 if η is a root node with no children then
 2     η.safeLiterals ← ∅
 3 else
 4     if η has only one child η′ then
 5         if η′ is marked as regularized then
 6             η.safeLiterals ← η′.safeLiterals ;
 7         else if η is left parent of η′ then
 8             η.safeLiterals ← η′.safeLiterals ∪ { η′.rightResolvedLiteral } ;
 9         else if η is right parent of η′ then
10             η.safeLiterals ← η′.safeLiterals ∪ { η′.leftResolvedLiteral } ;
11     else
12         η.safeLiterals ← ∅
```

Algorithm 4: `setSafeLiterals` for

And being HashSets, membership checking is done in constant time in the average case, and set intersection (line 12) can be done in $O(k.s)$, where k is the number of sets and s is the size of the smallest set.

Example 4.

5 Experiments

A prototype[1] of a (two-traversal) version of `GreedyLinearFirstOrderLowerUnits` has been implemented in the functional programming language Scala[2] as part of the Skeptik library[3].

Before evaluating this algorithm, we first generated several benchmark proofs. This was done by executing the SPASS[4] theorem prover on 2280 real first-order problems without equality of the TPTP Problem Library [5] (among them, 1032 problems are known to be unsatisfiable). In order to generate pure resolution proofs, most advanced inference rules used by SPASS were disabled. The Euler Cluster at the University of Victoria[6] was used and the time limit was 300 seconds per problem. Under these conditions, SPASS was able to generate 308 proofs.

The evaluation of `GreedyLinearFirstOrderLowerUnits` was performed on a laptop (2.8GHz Intel Core i7 processor with 4 GB of RAM (1333MHz DDR3)

[1] Source code available at https://github.com/jgorzny/Skeptik
[2] http://www.scala-lang.org/
[3] https://github.com/Paradoxika/Skeptik
[4] http://www.spass-prover.org/
[5] http://www.cs.miami.edu/~tptp/
[6] https://rcf.uvic.ca/euler.php

available to the Java Virtual Machine). For each benchmark proof ψ, we measured[7] the time needed to compress the proof ($t(\psi)$) and the compression ratio ($(|\psi| - |\alpha(\psi)|)/|\psi|$), where $|\psi|$ is the length of ψ (i.e. the number of axioms, resolution and contractions (ignoring substitutions)) and $\alpha(\psi)$ is the result of applying `GreedyLinearFirstOrderLowerUnits` to ψ.

The proofs generated by SPASS were small (with lengths from 3 to 49). These proofs are specially small in comparison with the typical proofs generated by SAT- and SMT-solvers, which usually have from a few hundred to a few million nodes. The number of proofs (compressed and uncompressed) per length is shown in Figure 1 (b). Uncompressed proofs are those which had either no lowerable units to lower or for which `GreedyLinearFirstOrderLowerUnits` failed and returned the original proof. Such failures occurred on only 14 benchmark proofs. Among the smallest of the 308 proofs, very few proofs were compressed. This is to be expected, since the likelihood that a very short proof contain a lowerable unit (or even merely a unit with more than one child) is low. The proportion of compressed proofs among longer proofs is, as expected, larger, since they have more nodes and it is more likely that some of these nodes are lowerable units. 13 out of 18 proofs with length greater than or equal to 30 were compressed.

Figure 1 (a) shows a box-whisker plot of compression ratio with proofs grouped by length and whiskers indicating minimum and maximum compression ratio achieved within the group. Besides the median compression ratio (the horizontal thick black line), the chart also shows the mean compression ratios for all proofs of that length and for all compressed proofs (the red cross and the blue circle). In the longer proofs (length greater than 34), the median and the means are in the range from 5% to 15%, which is satisfactory in comparison with the total compression ratio of 7.5% that has been measured for the propositional `LowerUnits` algorithm on much longer propositional proofs [3].

Figure 1 (c) shows a scatter plot comparing the length of the input proof against the length of the compressed proof. For the longer proofs (circles in the right half of the plot), it is often the case that the length of the compressed proof is significantly lesser than the length of the input proof.

Figure 1 (d) plots the cumulative original and compressed lengths of all benchmark proofs (for an x-axis value of k, the cumulative curves show the sum of the lengths of the shortest k input proofs). The total cumulative length of all original proofs is 4429 while the cumulative length of all proofs after compression is 3929. This results in a total compression ratio of 11.3%, which is impressive, considering that the inclusion of all the short proofs (in which the presence of lowerable units is a priori unlikely) tends to decrease the total compression ratio. For comparison, the total compression ratio considering only the 100 longest input proofs is 18.4%.

Figure 1 also indicates an interesting potential trend. The gap between the two cumulative curves seems to grow superlinearly. If this trend is extrapolated, progressively larger compression ratios can be expected for longer proofs. This

[7] The raw data is available at https://docs.google.com/spreadsheets/d/1F1-t2OuhypmTQhLU6yTj42aiZ5CqqaZvhVvOzeFgn0k/

is compatible with Theorem 10 in [7], which shows that, for proofs generated by eagerly resolving units against all clauses, the propositional `LowerUnits` algorithm can achieve quadratic assymptotic compression. SAT- and SMT-solvers based on CDCL (Conflict-Driven Clause Learning) avoid eagerly resolving unit clauses by dealing with unit clauses via boolean propagation on a conflict graph and extracting subproofs from the conflict graph with every unit being used at most once per subproof (even when it was used multiple times in the conflict graph). Saturation-based automated theorem provers, on the other hand, might be susceptible to the eager unit resolution redundancy described in Theorem 10 [7]. This potential trend would need to be confirmed by further experiments with more data (more proofs and longer proofs).

The total time needed by SPASS to solve the 308 problems for which proofs were generated was 2403 seconds, or approximately 40 minutes (running on the Euler Cluster and including parsing time and proof generation time for each problem). The total time for `GreedyLinearFirstOrderLowerUnits` to be executed on all 308 proofs was just under 5 seconds on a simple laptop (including parsing each proof). Therefore, `GreedyLinearFirstOrderLowerUnits` is a fast algorithm. For a very small overhead in time (in comparison to proving time), it may simplify the proof considerably.

6 Conclusions and Future Work

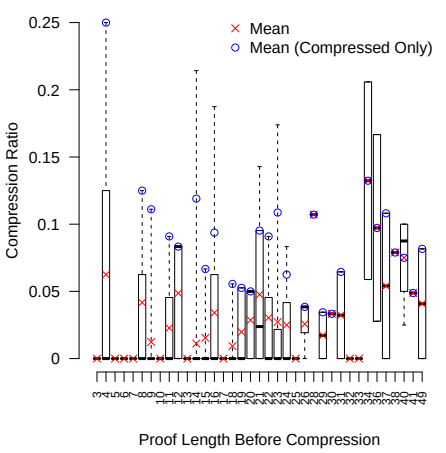
`GreedyLinearFirstOrderLowerUnits` is our first attempt to lift a propositional proof compression algorithm to the first-order case. We consider this algorithm a prototype, useful to evaluate whether this approach is promising. The experimental results discussed in the previous section are encouraging, especially in comparison with existing results for the propositional case. In the near future, we shall seek improvements of this algorithm as well as other ways to overcome the complexity and the bookkeeping difficulties of `FirstOrderLowerUnits`.

The difficulties related to unit reintroduction suggest that other propositional proof compression algorithms that do not require reintroduction (e.g. `RecyclePivotsWithIntersection` [7]) might need less sophisticated bookkeeping when lifted to first-order.

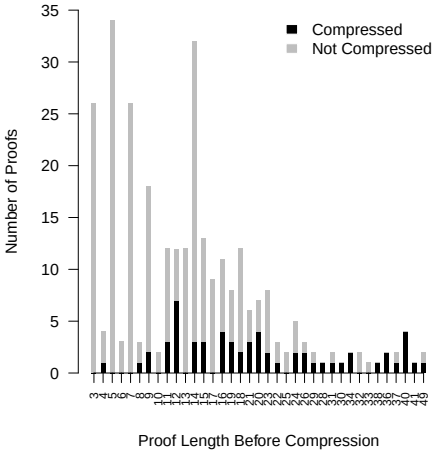
The efficiency and versatility of contemporary automated theorem provers depend on inference rules (e.g. equality rules) and techniques (e.g. splitting) that go beyond the pure resolution calculus. The eventual generalization of the compression algorithms to support such extended calculi will be essential for their usability on a wider range of problems.

References

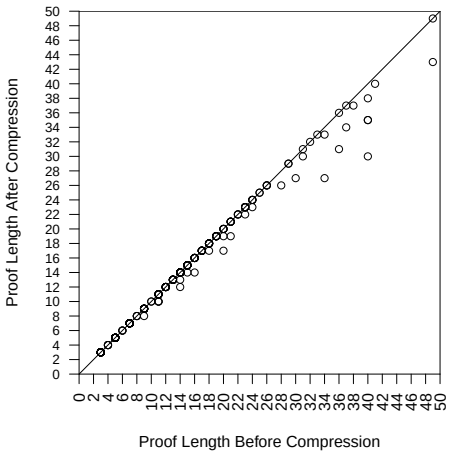
1. Hasan Amjad. Compressing propositional refutations. *Electr. Notes Theor. Comput. Sci.*, 185:3–15, 2007.
2. Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In Hana Chockler and Alan J.



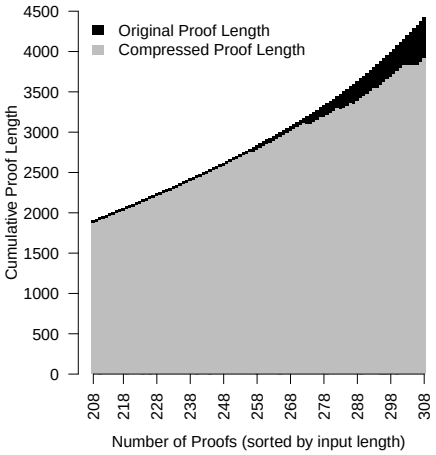
(a) Compression ratio



(b) Number of (non-)compressed proofs



(c) Compressed length against input length



(d) Cumulative proof lengths

Fig. 1: Experimental results

Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2008.
3. Joseph Boudou and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs by lowering subproofs. In Didier Galmiche and Dominique Larchey-Wendling, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 22th International Conference, TABLEAUX 2013, Nancy, France, September 16-19, 2013. Proceedings*, volume 8123 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2013.
4. Edmund M. Clarke and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*. Springer, 2010.
5. Scott Cotton. Two techniques for minimizing resolution proofs. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 306–312. Springer, 2010.
6. Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploring and exploiting algebraic and graphical properties of resolution. In *8th International Workshop on Satisfiability Modulo Theories - SMT 2010*, Edinburgh, Royaume-Uni, July 2010.
7. Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2011.
8. Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. Algorithmic introduction of quantified cuts. *Theor. Comput. Sci.*, 549:1–16, 2014.
9. Bruno Woltzenlogel Paleo. Atomic cut introduction by resolution: Proof structuring and compression. In Clarke and Voronkov [4], pages 463–480.
10. Simone Fulvio Rollini, Roberto Bruttomesso, and Natasha Sharygina. An efficient and flexible approach to resolution proof reduction. In Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software: Verification and Testing*, volume 6504 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2011.
11. Carsten Sinz. Compressing propositional proofs by common subproof extraction. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *EUROCAST*, volume 4739 of *Lecture Notes in Computer Science*, pages 547–555. Springer, 2007.
12. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
13. Jirí Vyskocil, David Stanovský, and Josef Urban. Automated proof compression by invention of new definitions. In Clarke and Voronkov [4], pages 447–462.