Partial Regularization of First-Order Resolution Proofs

Jan Gorzny[1] * and Bruno Woltzenlogel Paleo[2,3]

[1] jgorzny@uwaterloo.ca, University of Waterloo, Canada
[2] bruno@logic.at, Vienna University of Technology, Austria
[3] Australian National University

Abstract. This paper describes the generalization of the proof compression algorithm `RecyclePivotsWithIntersection` from propositional to first-order logic. The generalized algorithm performs partial regularization of resolution proofs containing resolution and factoring inferences with *unification*, as generated by many automated theorem provers. An empirical evaluation of the generalized algorithm and its combinations with `GreedyLinearFirstOrderLowerUnits` is also presented.

1 Introduction

First-order automated theorem provers, commonly based on resolution and superposition calculi, have recently achieved a high degree of maturity. Proof production is a key feature that has been gaining importance, since proofs are crucial for applications that require certification of a prover's answers or information extractable from proofs (e.g. unsat cores, interpolants, instances of quantified variables). Nevertheless, proof production is non-trivial [11], and the best, most efficient provers do not necessarily generate the best, least redundant proofs.

For proofs using propositional resolution generated by SAT- and SMT-solvers, there is a wide variety of proof compression techniques. Algebraic properties of the resolution operation that might be useful for compression were investigated in [5]. Compression algorithms based on rearranging and sharing chains of resolution inferences have been developed in [2] and [12]. Cotton [4] proposed an algorithm that compresses a refutation by repeatedly splitting it into a proof of a heuristically chosen literal ℓ and a proof of $\overline{\ell}$, and then resolving them to form a 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 [3] and improved in [6].

In contrast, there has been much less work on simplifying first-order proofs. For tree-like sequent calculus proofs, algorithms based on cut-introduction [9, 8] have been proposed. However, converting a DAG-like resolution or superposition proof, as usually generated by current provers, into a tree-like sequent calculus proof may increase the size of the proof. For arbitrary proofs in the TPTP

* Supported by the Google Summer of Code 2014 program.

[13] format (including DAG-like first-order resolution proofs), there is a simple algorithm [15] that looks for terms that occur often in any TSTP [13] proof and introduces abbreviations for these terms.

The work reported in this paper is part of a new trend that aims at lifting successful propositional proof compression algorithms to first-order logic. Our first target was the propositional `LowerUnits` algorithm, which delays resolution steps with unit clauses, resulting in the `GreedyLinearFirstOrderLowerUnits` (`GFOLU`) algorithm [7]. Here we continue this line of research by lifting the `RecyclePivotsWithIntersection` (`RPI`) algorithm [6], which is an improvement of the `RecyclePivots` (`RP`) algorithm [3], providing better compression on proofs where nodes have several children.

Section 2 introduces the first-order resolution calculus and the notations used in this paper. Section 4 discusses the challenges that arise in the first-order case (mainly due to unification), which are not present the propositional case. Section 5 describes an algorithm that overcomes these challenges. Section 6 presents experimental results obtained by applying this algorithm, and its combinations with `GFOLU`, on hundreds of proofs generated with the SPASS theorem prover. Section 7 concludes the paper.

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. □

3 The Propositional Algorithm

`RPI` (formally defined in Appendix A) removes *irregularities*, which are resolution inferences with a node η when the resolved literal (a.k.a. *pivot*) occurs as the pivot of another inference located below in the path from η to the root of the proof. In the worst case, regular resolution proofs can be exponentially bigger than irregular ones, but `RPI` takes care of regularizing the proof only partially, removing inferences only when this does not enlarge the proof.

`RPI` traverses the proof twice. On the first traversal (bottom-up), it stores for each node a set of *safe literals* that are resolved in all paths below it in the proof or that occur in the root clause of the proof. 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 the parent containing the safe literal. To do this replacement efficiently, the replacement is postponed by marking the other parent as a `deletedNode`. Then, on a single second traversal (top-down), regularization

is performed: any node that has a parent node marked as a `deletedNode` is replaced by its other parent.

The `RPI` and the `RP` algorithms differ from each other mainly in the computation of the safe literals of a node that has many children. While the former returns the intersection as shown in Algorithm 6, the latter returns the empty set. Moreover, while in `RPI` the safe literals of the root node contain all the literals of the root clause, in `RP` the root node is always assigned an empty set of literals.

4 First-Order Challenges

In this section, we describe challenges that have to be overcome in order to successfully adapt `RPI` 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 taken into account in a naive way.

Example 1. Consider the following proof ψ. When computed as in the propositional case, the safe literals for η_3 are $\{\vdash q(c),\ p(a, X)\}$.

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

As neither of η_3's pivots is syntactically equal to a safe literal, the propositional `RPI` algorithm would not change ψ. However, η_3's left pivot $p(W, X) \in \eta_1$ is unifiable with the safe literal $p(a, X)$. Regularizing η_3, by deleting the edge between η_2 and η_3 and replacing η_3 by η_1, leads to further deletion of η_4 (because it is not resolvable with η_1) and finally to the much shorter proof below.

$$\dfrac{\eta_1\colon\ \vdash p(W, X) \qquad \eta_6\colon\ p(Y, b)\ \vdash}{\psi'\colon\ \bot}$$

Unlike in the propositional case, where a pivot must be syntactically equal to a safe literal for regularization to be possible, the example above suggests that, in the first-order case, it might suffice that a pivot be unifiable with a safe literal. However, there are cases, as shown in the example below, where mere unifiability is not enough and greater care is needed.

Example 2. Again, the safe literals for η_3, when computed as in the propositional case, are $\{\vdash q(c),\ p(a, X)\}$, and as the pivot $p(a, c)$ is unifiable with the safe literal $p(a, X)$, η_3 appears to be a candidate 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 require resolution between η_1 and η_6, which is not possible.

One way to prevent the problem depicted above would be to require the pivot to be not only unifiable but in fact more general than a safe literal. A weaker (and better) requirement is possible, however, as defined below.

Definition 2. *Let η be a node with pivot ℓ' unifiable with 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 way to ensure this property is met is to slightly modify the notion of safe literals, by applying the unifier of the resolution step to the each pivot before adding it to the safe literals (cf. algorithm 3, lines 8 and 10). In the case of Example 2, this would result in η_3 having the safe literals $\{\vdash q(c),\ p(a,b)\}$, where clearly the pivot $p(a,c)$ in η_1 is not safe.

Example 3. Satisfying the pre-regularization unifiability property is not sufficient. Consider the proof ψ below. After collecting the safe literals, η_3's safe literals are $\{q(T,V), p(c,d) \vdash q(f(a,e),c)\}$.

$$\dfrac{\dfrac{\eta_1\colon p(U,V) \vdash q(f(a,V),U) \qquad \eta_2\colon q(f(a,X),Y), q(T,X) \vdash q(f(a,Z),Y)}{\eta_3\colon p(U,V), q(T,V) \vdash q(f(a,Z),U)} \qquad \eta_4\colon \vdash q(R,S)}{\dfrac{\eta_6\colon \vdash p(c,d) \qquad \eta_5\colon p(U,V) \vdash q(f(a,Z),U)}{\eta_8\colon q(f(a,e),c) \vdash \qquad \eta_7\colon \vdash q(f(a,Z),c)}}$$
$$\psi\colon \bot$$

η_3's pivot $q(f(a,V),U)$ is unifiable to (and even more general than) the safe literal $q(f(a,e),c)$. Attempting to regularize η_3 would lead to the removal of η_2, the replacement of η_3 by η_1 and the removal of η_4 (because η_1 does not contain the pivot required by η_5), with η_5 also being replaced by η_1. Then resolution between η_1 and η_6 results in η_7', which cannot be resolved with η_8, as shown below.

$$\dfrac{\eta_8\colon q(f(a,e),c) \vdash \qquad \dfrac{\eta_6\colon \vdash p(c,d) \qquad \eta_1\colon p(U,V) \vdash q(f(a,V),U)}{\eta_7'\colon \vdash q(f(a,d),c)}}{\psi'\colon ??}$$

η_1's literal $q(f(a,V),U)$, which would be resolved with η_8's literal, was changed to $Q(f(a,d),c)$ due to the resolution between η_1 and η_6.

Thus we additionally require the following property be satisfied.

Definition 3. *Let η be a node with safe literals ϕ that is marked for regularization with parents η_1 and η_2, where η_2 is marked as a* `deletedNode` *in a proof ψ. η is said to satisfy the* regularization unifiability property *in ψ if there exists a substitution σ such that $\eta_1\sigma \subseteq \phi$.*

This property ensures that the remainder of the proof does not expect a variable in η_1 to be unified to different values simultaneously. This property is not necessary in the propositional case, as the replacement node would not change lower in the proof.

input : A first-order proof ψ
output: A possibly less-irregular first-order 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 $\psi' \leftarrow \mathrm{fix}(\psi')$;
7 **return** ψ';

Algorithm 1: `FORPI`

input : A node $\psi = \psi_L \odot^{\sigma_L\sigma_R}_{\ell_L\ell_R} \psi_R$
output: nothing (but the proof containing ψ may be changed)

1 **if** $\exists\sigma$ *and* $\ell \in \psi$.safeLiterals *such that* $\ell\sigma = \ell_R$ *or* $\ell = \ell_R\sigma$ **then**
2 **if** $\exists\sigma'$ *such that* $\psi_R\sigma' \subseteq \psi$.safeLiterals **then**
3 mark ψ_L as `deletedNode` ;
4 mark ψ as regularized
5 **else if** $\exists\sigma$ *and* $\ell \in \psi$.safeLiterals *such that* $\ell\sigma = \ell_L$ *or* $\ell = \ell_L\sigma$ **then**
6 **if** $\exists\sigma'$ *such that* $\psi_L\sigma' \subseteq \psi$.safeLiterals **then**
7 mark ψ_R as `deletedNode` ;
8 mark ψ as regularized

Algorithm 2: `FOregularizeIfPossible`

5 First-Order RecyclePivotsWithIntersection

This section presents `FirstOrderRecyclePivotsWithIntersection` (`FORPI`), Algorithm 1, a first-order generalization of `RPI`. `FORPI` traverses the proof in a bottom-up manner, storing for every node a set of safe literals. The set of safe literals for a node ψ is computed from the set of safe literals of its children (cf. Algorithm 3), similarly to the propositional case, but additionally applying unifiers to the resolved pivots (cf. Example 2). If one of the node's resolved literals can be unified to a literal in the set of safe literals, then it may be possible to regularize the node by replacing it by one of its parents.

In the first-order case, we additionally check for the regularization property (cf. lines 2 and 6 of Algorithm 2). Similarly to `RPI`, instead of replacing the irregular node by one of its parents immediately, its other parent is marked as a `deletedNode`, as shown in Algorithm 2. As in the propositional case, fixing of the proof is postponed to another (single) traversal, as regularization proceeds top-down and only nodes below a regularized node may require fixing. During fixing, the irregular node is actually replaced by the parent that is not marked as `deletedNode`. During proof fixing, factoring inferences can be applied, in order to compress the proof further.

input : A first-order resolution node ψ
output: nothing (but the node ψ gets a set of safe literals)

if ψ *is a root node with no children* **then**
  ψ.safeLiterals $\leftarrow$ ψ.clause
else
  foreach $\psi' \in \psi$.children **do**
    if ψ' *is marked as regularized* **then**
      safeLiteralsFrom(ψ') $\leftarrow$ ψ'.safeLiterals ;
    else if $\psi' = \psi \odot_{\ell_L \ell_R}^{\sigma_L \sigma_R} \psi_R$ *for some* ψ_R **then**
      safeLiteralsFrom(ψ') $\leftarrow$ ψ'.safeLiterals $\cup$ $\{ \ell_R \sigma_R \}$
    else if $\psi' = \psi_L \odot_{\ell_L \ell_R}^{\sigma_L \sigma_R} \psi$ *for some* ψ_L **then**
      safeLiteralsFrom(ψ') $\leftarrow$ ψ'.safeLiterals $\cup$ $\{ \ell_L \sigma_L \}$
  ψ.safeLiterals $\leftarrow \bigcap_{\psi' \in \psi.\text{children}}$ safeLiteralsFrom(ψ')

Algorithm 3: `FOsetSafeLiterals`

6 Experiments

A prototype[1] version of `FORPI` has been implemented in the functional programming language Scala as part of the Skeptik library. This library includes an implementation of `GFOLU` [7]. In order to evaluate the algorithm's effectiveness, `FORPI` was tested on two data sets: proofs generated by a real theorem prover and artificial proofs resolution proofs. The data is included in the source code repository.

First, `FORPI` was evaluated on the same set of proofs used to evaluate `GFOLU`. This data was generated by executing the SPASS (`http://www.spass-prover.org/`) theorem prover on 2280 real first-order problems without equality of the TPTP Problem Library (among them, 1032 problems are known to be unsatisfiable). In order to generate pure resolution proofs, the advanced inference rules of SPASS were disabled. The proofs were originally generated on the Euler Cluster at the University of Victoria with a time limit of 300 seconds per problem. Under these conditions, SPASS generated 308 proofs. The proofs generated by SPASS were small: proof lengths varied from 3 to 49, and the number of resolutions in a proof ranged from 1 to 32.

In order to test `FORPI`'s effectiveness on larger proofs, randomly generated first-order resolution proofs were also used. Proofs were generated by In order to match the maximum number of potential proofs from the TPTP Problem Library, 2280 randomly generated proofs were used as the second data set. The proofs generated by were much larger than those of the first data set: proof lengths varied from 95 to 700, while the number of resolutions in a proof ranged from 48 to 368.

In order to maintain consistency, the same system and metrics were used. Proof compression and proof generation was performed on a laptop (2.8GHz Intel Core i7 processor with 4GB of RAM (1333MHz DDR3) available to the

[1] Source code available at `https://github.com/jgorzny/Skeptik`

Java Virtual Machine); proofs from the first data set were not generated again. For each proof ψ, we measured the time needed to compress the proof ($t(\psi)$) and the compression ratio in terms of resolutions (($|\psi| - |\alpha(\psi)|)/|\psi|$) where $|\psi|$ is the number of resolutions in the proof, and $\alpha(\psi)$ is the result of applying a compression algorithm or the composition of `FORPI` and `GFOLU`.

Only nine proofs from the TPTP data set were compressed by `FORPI`, reducing the number of resolutions by at least one but at most three. However, X% of the randomly generated proofs achieved some compression using `FORPI`. Figure 1 shows a box-whisker plot of compression ratio with proofs grouped by number of resolutions and whiskers indicating a minimum and maximum compression ratio achieved within the group. The circles indicate the mean compression ratio when considering only proofs for which some compression was achieve, and these points indicate that when larger proofs can be compressed, they are compressed more than smaller proofs. `FORPI` was able to achieve an overall average compression ratio of X% on this data; for comparison, the propositional `RPI` compression ratio is X%.

Figure 2 (a) shows the number of proofs (compressed and uncompressed) per grouping based on number of resolutions in the proof. The red (respectively dark gray) data shows the number of compressed (respectively, uncompressed) proofs for the TPTP data set, while the green (respectively light gray) data shows the number of compressed (respectively, uncompressed) proofs for the random proofs. Darker colours indicate the number of proofs compressed by both `FORPI` and `GFOLU` and both compositions of these algorithms; lighter colours indicate cases were `FORPI` succeeded, but at least one of `GFOLU` or a combination of these algorithms achieved no compression. The random proofs indicate that many proofs have some irregularity, but this is not reflected in the TPTP data; this is unsurprising given the size of the proofs in the TPTP data set.

Figure 2 (b) shows a scatter plot comparing the number resolutions of the input proof against the number of resolutions in the compressed proof. The data resulting from the TPTP data is magnified by the internal plot. For the randomly generated proofs (points to the left of the blown-up area), it is often the case that the length of the compressed proof is significantly lesser than the length of the input proof. In some cases, `FORPI` is able to reduce the number of resolutions from the hundreds to single digit values. Interestingly, `GFOLU` appears to reduce the the number of resolutions by a linear factor. We conjecture that this is due to the method by which the data was generated: specifically, ... TODO

Figure 2 (c) shows a scatter plot comparing the difference in the cumulative number resolutions of the first x input proofs against the cumulative number of resolutions in the first x compressed proofs. Again, the TPTP data is magnified in the sub-plot. Where, we see that the use of both algorithms is always better than using a single algorithm, and further that applying `FORPI` after `GFOLU` appears to be the best option.

Figure 2 (d) shows a scatter plot comparing the size of compression obtained by applying `FORPI` before `GFOLU` versus `GFOLU` before `FORPI`. Data generated from the TPTP data set is marked in red; the rest are obtained from random proofs.

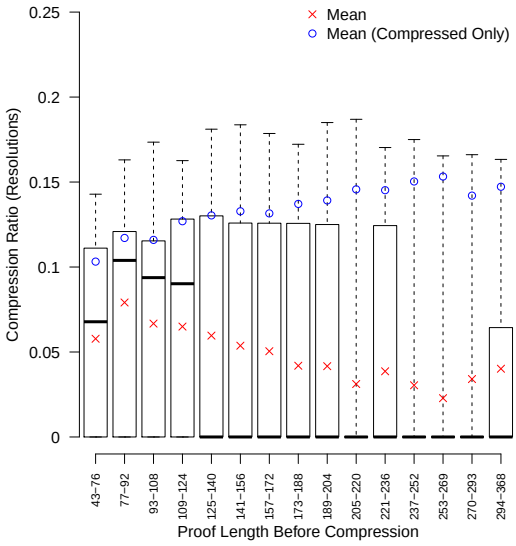


Fig. 1: Compression Ratio (`FORPI`)

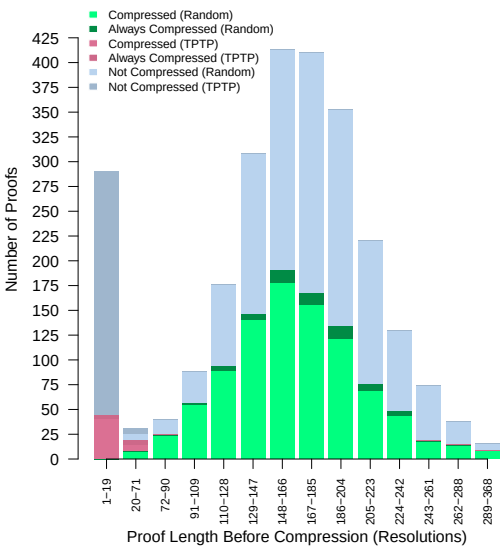
Points that lie on the line $y = x$ have the same size after both combinations. The data confirms(denies) the suggestion from Figure 2 (c) that `FORPI` should be applied after `GFOLU`: X points lie beneath the line, while X are above the line to maximize the likelihood of compression. However, the applying `FORPI` first may result in greater compression.

SPASS required approximately 40 minutes (running on a cluster and including proof generation time for each problem) to solve the generated proofs. The total time for `GFOLU` and `FORPI` to be executed on all 308 TPTP proofs was just under X seconds on a simple laptop. The random proofs were generated in X seconds, and took approximately X minutes to compress, both measured on a laptop. All times including parsing time. These compression algorithms continue to be very fast, and may simplify the proof considerably for a relatively quick time cost.

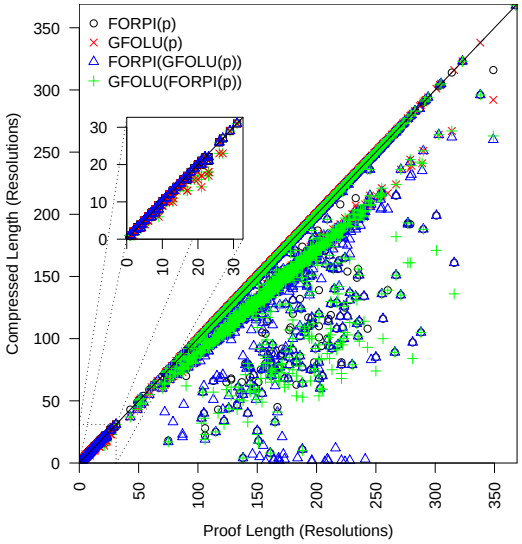
7 Conclusions and Future Work

The main contribution of this paper is the generalization of the propositional proof compression algorithm `RPI` to the first-order case. As indicated in Section 4, the generalization is challenging, because unification changes the pivots and, consequently, must be taken into account when collecting safe literals and marking nodes for deletion.

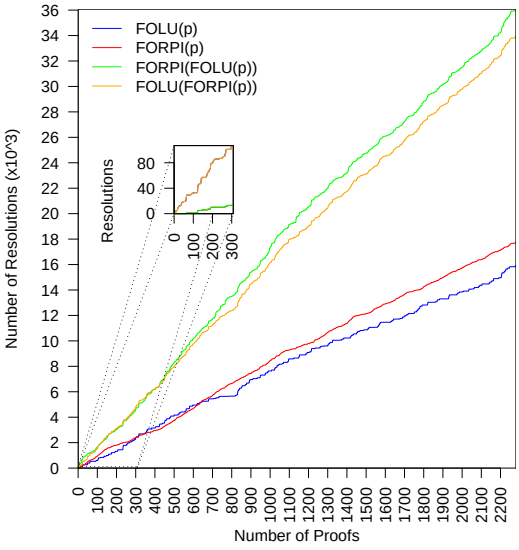
Every computational experiment evaluates not only the algorithm but also the data on which it is executed. Although the experimental results are not as promissing as expected, this is due to the fact that the 308 proofs currently



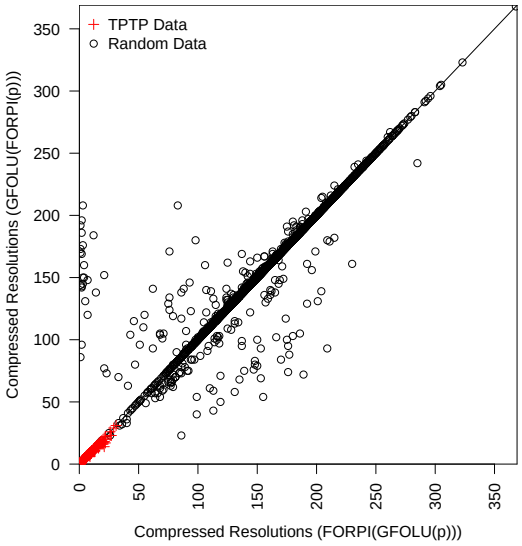
(a) Number of (non-)compressed proofs



(b) Compressed length against input length



(c) Cumulative proof compression



(d) `FORPI` (`GFOLU` (p)) vs. `GFOLU` (`FORPI` (p))

Fig. 2: `GFOLU` & `FORPI` Combination Results

available are too short to contain a significant amount of irregularities. This is a valuable piece of information, allowing us to conclude that it is not worth applying `FORPI` to pure resolution proofs which current state-of-the-art first-order theorem provers seem capable of producing. Nevertheless, based on our positive results for `RPI` on much longer proofs generated by SAT and SMT solvers [6], `FORPI` remains a promising option to be revisited in the future, when the performance of first-order theorem provers catch up with advances in SAT and SMT and taller first-order benchmark proofs become available.

References

1. *LPAR 16th Intl. Conf., Dakar, Senegal, Rev. Sel. Papers*, LNCS. Springer, 2010.
2. H. Amjad. Compressing propositional refutations. *Electr. Notes Theor. Comput. Sci.*, 185:3–15, 2007.
3. O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-time reductions of resolution proofs. In *Haifa Verif. Conf.*, LNCS, pages 114–128. Springer, 2008.
4. S. Cotton. Two techniques for minimizing resolution proofs. In Ofer Strichman and Stefan Szeider, editors, *SAT 2010*, LNCS, pages 306–312. Springer, 2010.
5. P. Fontaine, S. Merz, and B. Woltzenlogel Paleo. Exploring and exploiting algebraic and graphical properties of resolution. In *8th Intl. Wkshp. on SMT*, 2010.
6. P. Fontaine, S. Merz, and B. Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In *CADE*, LNCS, pages 237–251. Springer, 2011.
7. J. Gorzny and B. Woltzenlogel Paleo. Towards the compression of first-order resolution proofs by lowering unit clauses. In *CADE*, 2015.
8. S. Hetzl, A. Leitsch, G. Reis, and D. Weller. Algorithmic introduction of quantified cuts. *Theor. Comput. Sci.*, 549:1–16, 2014.
9. B. Woltzenlogel Paleo. Atomic cut introduction by resolution: Proof structuring and compression. In *LPAR-16* [1], pages 463–480.
10. S. F. Rollini, R. Bruttomesso, and N. Sharygina. An efficient and flexible approach to resolution proof reduction. In *Hardware and Software: Verification and Testing*, LNCS, pages 182–196. Springer, 2011.
11. S. Schulz and G. Sutcliffe. Proof generation for saturating first-order theorem provers. In D. Delahaye and B. Woltzenlogel Paleo, editors, *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*. College Publications, London, UK, 2015.
12. C. Sinz. Compressing propositional proofs by common subproof extraction. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *EUROCAST*, LNCS, pages 547–555. Springer, 2007.
13. 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.
14. G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*. Springer-Verlag, 1983.
15. J. Vyskocil, D. Stanovský, and J. Urban. Automated proof compression by invention of new definitions. In *LPAR* [1], pages 447–462.

A Algorithm `RecyclePivotsWithIntersection`

Note: for the reviewers' convenience, this appendix summarizes [6].

`RecyclePivotsWithIntersection` (`RPI`) [6] aims at compressing irregular proofs. It can be seen as a simple but significant modification of the `RP` algorithm described in [3], from which it derives its name. Although in the worst case full regularization can increase the proof length exponentially [14], these algorithms show that many irregular proofs can have their length decreased if a careful partial regularization is performed.

Consider an irregular proof of the form $\psi[\eta \odot_p \psi'[\eta' \odot_p \eta'']]$ and assume, without loss of generality, that $p \in \eta$ and $p \in \eta'$. Then, if $\eta' \odot_p \eta''$ is replaced by η'' within the proof-context $\psi'[\,]$, the clause $\eta \odot_p \psi'[\eta'']$ subsumes the clause $\eta \odot_p \psi'[\eta' \odot_p \eta'']$, because even though the literal $\neg p$ of η'' is propagated down, it gets resolved against the literal p of η later on below in the proof. More precisely, even though it might be the case that $\neg p \in \psi'[\eta'']$ while $\neg p \notin \psi'[\eta' \odot_p \eta'']$, it is necessarily the case that $\neg p \notin \eta \odot_p \psi'[\eta' \odot_p \eta'']$ and $\neg p \notin \eta \odot_p \psi'[\eta'']$.

Although the remarks above suggest that it is safe to replace $\eta' \odot_p \eta''$ by η'' within the proof-context $\psi'[\,]$, this is not always the case. If a node in $\psi'[\,]$ has a child in $\psi[\,]$, then the literal $\neg p$ might be propagated down to the root of the proof, and hence, the clause $\psi[\eta \odot_p \psi'[\eta'']]$ might not subsume the clause $\psi[\eta \odot_p \psi'[\eta' \odot_p \eta'']]$. Therefore, it is only safe to do the replacement if the literal $\neg p$ gets resolved in all paths from η'' to the root or if it already occurs in the root clause of the original proof $\psi[\eta \odot_p \psi'[\eta' \odot_p \eta'']]$.

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 4).

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

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 4: `RPI`

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 marked as `deletedNode`, as shown in Algorithm 5. Only later during fixing, the irregular node is actually replaced by its surviving parent (i.e. the parent that is not marked as `deletedNode`).

The set of safe literals of a node η can be computed from the set of safe literals of its children (cf. Algorithm 6). 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 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 `RP` 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 6, `RP` returns the empty set (cf. Algorithm 7). Additionally, while in `RPI` the safe literals of the root node contain all the literals of the root clause, in `RP` 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 6 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. 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.

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

Algorithm 5: `regularizeIfPossible`

```
   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 6: `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 7: `setSafeLiterals` for `RP`