Compression of Propositional Resolution Proofs via Partial Regularization[*]

Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo

University of Nancy and INRIA, Nancy, France
{Pascal.Fontaine,Stephan.Merz,Bruno.WoltzenlogelPaleo}@inria.fr

Abstract. This paper describes two algorithms for the compression of propositional resolution proofs. The first algorithm, `RecyclePivotsWithIntersection`, performs partial regularization, removing an inference η when it is redundant in the sense that its pivot literal already occurs as the pivot of another inference located below in the path from η to the root of the proof. The second algorithm, `LowerUnits`, delays the resolution of (both input and derived) unit clauses, thus removing (some) inferences having the same pivot but possibly occurring also in different branches of the proof.

1 Introduction

Propositional satisfiability (SAT) solving has made enormous progress during the recent decade, and powerful decision procedures are being used in many different contexts, such as hardware and software verification, knowledge representation, and diagnostic applications (see [3] for a thorough presentation of techniques and applications of SAT solving). SAT solving also forms the backbone for automated reasoning over more expressive logics, such as in SMT (satisfiability modulo theories) solving (see [2] for a detailed account of techniques used in SMT solvers). For many applications, it is not enough to just obtain a yes/no answer to the decision problem, but one is also interested in a justification of the verdict, that is, a model satisfying the original formula, or a proof showing that no such model exists. For example, in the context of proof carrying code [7], the code producer must provide a proof that will be checked by the code consumer. In the context of SAT solving, it is well understood how decision procedures can be adapted to construct a resolution proof while performing proof search. However, proofs output by SAT solvers can be huge (millions of learned clauses and tens or hundreds of megabytes for typical benchmark cases), and techniques for obtaining small proofs become of interest. Producing a proof of minimum size is an NP-hard problem, so it is important to find heuristics of low (preferably linear) complexity that achieve interesting reductions in practical cases. Going beyond trivial optimizations, such as eliminating inferences that do not contribute to the final conclusion, one frequently observes that the same clause (or the same pivot literal) is used more than once within a proof, and even along

[*] This work was partly supported by the ANR DeCert project.

a single branch in a proof. Although it is not a priori the case that multiple uses of a clause (or pivot) are actually redundant or that their elimination results in a shorter proof, we concentrate in this work on identifying such cases and on corresponding transformations of proofs. Our algorithms are generalizations of similar techniques proposed in the literature [1, 5, 9]. We show that our techniques yield provably better results than previous algorithms, while their implementation is no more complex. A more detailed comparison with existing work appears in Section 7. We have implemented our algorithms and we presented experimental validations on standard benchmarks in Section 6.

2 The Resolution Calculus

A *literal* is an atomic formula or a negated atomic formula. A *clause* is a set of literals, $\bot$ denotes the *empty clause*. We write $\overline{\ell}$ to denote the dual of ℓ and $|\ell|$ for the atom underlying the literal ℓ (i.e., $\overline{p} = \neg p$, $\overline{\neg p} = p$, and $|p| = |\neg p| = p$ for an atomic formula p).

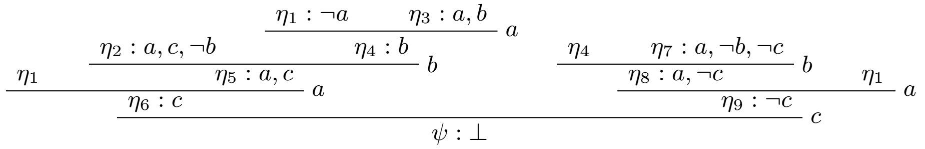
Definition 1. *A* resolution inference *is an instance of the following rule:*

$$\frac{\Gamma_1 \cup \{\ell\} \qquad \Gamma_2 \cup \{\overline{\ell}\}}{\Gamma_1 \cup \Gamma_2}\ |\ell|$$

The clauses $\Gamma_1 \cup \{\ell\}$ and $\Gamma_2 \cup \{\overline{\ell}\}$ are the inference's premises *and $\Gamma_1 \cup \Gamma_2$ (the* resolvent *of the premises) is its* conclusion*. The literal ℓ ($\overline{\ell}$) is the* left (right) resolved literal*, and $|\ell|$ is the* resolved atom *or* pivot*.* □

A *(resolution) proof* of a *clause* κ from a set of clauses C is a directed acyclic graph (dag): the *input nodes* are *axiom inferences* (without premises) whose conclusions are elements of C, the *resolvent nodes* are resolution inferences, and the proof has a node with conclusion κ. The dag contains an edge from a node η_1 to a node η_2 if and only if a premise of η_1 is the conclusion of η_2. In this case, η_1 is a *child* of η_2, and η_2 is a *parent* of η_1. A node with no children is a *root*. A *(resolution) refutation* of C is a resolution proof of $\bot$ from C. For the sake of brevity, given a node η, we say *clause* η or η*'s clause* meaning the conclusion clause of η, and *(sub)proof* η meaning the (sub)proof having η as its only root. The resolvent of κ_1 and κ_2 with pivot p can be denoted as $\kappa_1 \odot_p \kappa_2$. When the pivot is uniquely defined or irrelevant, we omit it and write simply $\kappa_1 \odot \kappa_2$. In this way, the set of clauses can be seen as an algebra with a commutative operator $\odot$ whose properties have been investigated in [6]; and terms in the corresponding term algebra denote resolution proofs in a notation style that is more compact and more convenient for describing resolution proofs than the usual graph notation.

Example 2. Consider the proof ψ shown below:



The node η_4 has pivot a, left (right) resolved literal $\neg a$ (a). Its conclusion is $\{b\}$ and its premises are the conclusions of its parents: the input nodes η_1 ($\{\neg a\}$) and η_3 ($\{a, b\}$). It has two children (η_5 and η_8). ψ can be compactly represented by the following proof term:

$$(\underbrace{\{\neg a\}}_{\eta_1} \odot (\{a, c, \neg b\} \odot \underbrace{(\eta_1 \odot \{a, b\})}_{\eta_4})) \odot ((\eta_4 \odot \{a, \neg b, \neg c\}) \odot \eta_1).$$

3 Redundant Proofs

A proof ψ of κ is considered *redundant* iff there exists another proof ψ' of κ' such that $\kappa' \subseteq \kappa$ (i.e. κ' subsumes κ) and $|\psi'| < |\psi|$ where $|\varphi|$ is the number of nodes in φ. The definition below describes two patterns of local redundancy: proofs matching them (modulo commutativity of $\odot$) can be easily compressed.

Definition 3 (Local Redundancy). *A proof containing a subproof of the shapes (here omitted pivots indicate that the resolvents must be uniquely defined)*

$$(\eta \odot \eta_1) \odot (\eta \odot \eta_2) \quad \textit{or} \quad \eta \odot (\eta_1 \odot (\eta \odot \eta_2))$$

is locally redundant. *Indeed, both of these subproofs can be equivalently replaced by the shorter subproof* $\eta \odot (\eta_1 \odot \eta_2)$. □

Example 4. The proofs below are two of the simplest examples of locally redundant proofs.

$$\dfrac{\dfrac{\eta : \neg a \qquad \eta_1 : a, b}{b}\,a \qquad \dfrac{\eta \qquad \eta_2 : a, \neg b}{\neg b}\,a}{\psi_1 : \bot}\,b$$

$$\dfrac{\eta : \neg a \qquad \dfrac{\eta_1 : a, b \qquad \dfrac{\eta \qquad \eta_2 : a, \neg b}{\neg b}\,a}{a}\,b}{\psi_2 : \bot}\,a$$

Both proofs can be rewritten to the shorter proof below:

$$\dfrac{\eta : \neg a \qquad \dfrac{\eta_1 : a, b \qquad \eta_2 : a, \neg b}{a}\,b}{\psi_3 : \bot}\,a$$

Note that, by locally permuting the lowermost inference with pivot a and the inference with pivot b, the proof ψ_2 can be transformed into ψ_1. This indicates that the two patterns given in Def. 3 can be seen as different manifestations of the same underlying phenomenon. They appear differently in resolution proofs because the resolution calculus enforces a sequential order of inferences even when the order actually does not matter. □

In the case of local redundancy, the pairs of redundant inferences having the same pivot occur close to each other in the proof. However, redundant inferences can also occur far apart in the proof. One could attempt to remove global redundancies by repeatedly permuting inferences until the redundant inferences appear next to each other. However this approach is intrinsically inefficient because many permutations must be considered and intermediate resolvents must be recomputed after every permutation.

The following definition generalizes Def. 3 by considering inferences with the same pivot that occur within different contexts. We write $\psi[\eta]$ to denote a *proof-context* $\psi[\_]$ with a single placeholder replaced by the subproof η.

Definition 5 ((Global) Redundancy). *A proof*

$$\psi[\psi_1[\eta \odot_p \eta_1] \odot \psi_2[\eta \odot_p \eta_2]] \quad \textit{or} \quad \psi[\psi_1[\eta \odot_p (\eta_1 \odot \psi_2[\eta \odot_p \eta_2])]]$$

is potentially (globally) redundant*. Furthermore, it is* (globally) redundant *if it can be rewritten to one of the following shorter proofs:*

$$\psi[\eta \odot_p (\psi_1[\eta_1] \odot \psi_2[\eta_2])] \quad \textit{or} \quad \eta \odot_p \psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]] \quad \textit{or} \quad \psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]].$$

Example 6. Consider the following proof ψ.

$$\dfrac{\dfrac{\dfrac{\eta : p, q \qquad \eta_1 : \neg p, r}{q, r}\,p \qquad \eta_3 : \neg q}{r}\,q \qquad \dfrac{\dfrac{\eta \qquad \eta_2 : \neg p, s, \neg r}{q, s, \neg r}\,p \qquad \eta_3}{s, \neg r}\,q}{\psi : s}\,r$$

It corresponds to the proof term $((\eta \odot_p \eta_1) \odot \eta_3) \odot ((\eta \odot_p \eta_2) \odot \eta_3)$, which is an instance of the first pattern appearing in Def. 5, hence ψ is potentially globally redundant. However, ψ is not globally redundant: the replacement terms according to Def. 5 contain $(\eta_1 \odot \eta_3) \odot (\eta_2 \odot \eta_3)$, which does not correspond to a proof. In particular, neither η_1 nor η_2 can be resolved with η_3, as they do not contain the literal q. □

The second pattern of potentially globally redundant proofs appearing in Def. 5 is related to the well-known notion of *regularity* [10]. Informally, a proof is irregular if there is a path from a node to the root of the proof such that a literal is used more than once as a pivot in this path.

Definition 7 (Irregularity). *A proof of the form* $\psi[\eta \odot_p (\eta_1 \odot \psi'[\eta' \odot_p \eta_2])]$ *is* irregular*.* □

The *regular resolution calculus* is the resolution calculus restricted so that irregular proofs are disallowed. Although it is still complete, it does not p-simulate the unrestricted resolution calculus [10]: there are unsatisfiable formulas whose shortest regular resolution refutations are exponentially longer than their shortest unrestricted resolution refutations.

4 Algorithm `LowerUnits`

A closer inspection of Example 6 shows that it relies on η's clause containing two literals: its literal q had to be resolved within the proof-contexts $\psi_1[\_]$ and $\psi_2[\_]$, and hence η could not be moved outside the contexts. It is easy to see that a potentially redundant proof is always redundant in case the redundant node contains a unit clause.

Theorem 8. *Let φ be a potentially redundant proof, and η be the redundant node. If η's clause is a unit clause, then φ is redundant.*

Proof. Consider first a proof of the form $\psi[\psi_1[\eta \odot \eta_1] \odot \psi_2[\eta \odot \eta_2]]$ and let ℓ be the only literal of η's clause. Then the clause $\psi_1[\eta_1] \odot \psi_2[\eta_2]$ contains the literal $\overline{\ell}$. Two cases can be distinguished, depending on whether the literal $\overline{\ell}$ gets propagated to the root of $\psi[\_]$:

1. In all paths from $\psi_1[\eta_1] \odot \psi_2[\eta_2]$ to the root of $\psi[\_]$, $\overline{\ell}$ gets resolved out: then, the clause $\psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]]$ is equal to the clause $\psi[\psi_1[\eta \odot \eta_1] \odot \psi_2[\eta \odot \eta_2]]$, and hence the original proof can be rewritten to $\psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]]$.
2. In some paths from $\psi_1[\eta_1] \odot \psi_2[\eta_2]$ to the root of $\psi[\_]$, $\overline{\ell}$ does not get resolved out: in this case, the clause of $\psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]]$ is equal to the clause of $\psi[\psi_1[\eta \odot \eta_1] \odot \psi_2[\eta \odot \eta_2]]$ with the additional literal $\overline{\ell}$. Consequently, the clause $\eta \odot (\psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]])$ is equal to the clause $\psi[\psi_1[\eta \odot \eta_1] \odot \psi_2[\eta \odot \eta_2]]$, and hence the original proof can be rewritten to $\eta \odot (\psi[\psi_1[\eta_1] \odot \psi_2[\eta_2]])$.

In both cases, since the rewriting to one of the three shorter proofs described in Definition 5 is possible, the proof is redundant. The case for potentially redundant proofs of the form $\psi[\psi_1[\eta \odot_p (\eta_1 \odot \psi_2[\eta \odot_p \eta_2])]]$ is analogous. □

The `LowerUnits` (`LU`) algorithm targets exactly the class of global redundancy stemming from multiple resolutions with unit clauses. The algorithm takes its name from the fact that, when this rewriting is done and the resulting proof is displayed as a dag, the unit node η appears lower (i.e., closer to the root) than it used to appear in the original proof.

A naive implementation exploiting Theorem 8 would require the proof to be traversed and fixed after each unit node is lowered. It is possible, however, to do better by first collecting and removing all the unit nodes in a single traversal, and afterwards fixing the whole proof in a single second traversal. Finally, the collected and fixed unit nodes have to be reinserted at the bottom of the proof (cf. Algorithms 1 and 2).

Care must be taken with cases when a unit node η' occurs above in the subproof that derives another unit node η. In such cases, η depends on η'. Let ℓ be the single literal of the unit clause of η'. Then any occurrence of $\overline{\ell}$ in the subproof above η will not be cancelled by resolution inferences with η' anymore. Consequently, $\overline{\ell}$ will be propagated downwards when the proof is fixed and will appear

```
input  : A proof $\psi$
output: A proof $\psi'$ with no global redundancy with unit redundant node
(unitsQueue, $\psi_b$) ← collectUnits($\psi$);
$\psi_f$ ← fix($\psi_b$);
fixedUnitsQueue ← fix(unitsQueue);
$\psi'$ ← reinsertUnits($\psi_f$, fixedUnitsQueue) ;
return $\psi'$;
```

Algorithm 1: `LowerUnits`

```
input  : A proof $\psi$
output: A pair containing a queue of all unit nodes (unitsQueue) that are used
        more than once in $\psi$ and a broken proof $\psi_b$
$\psi_b \leftarrow \psi$;
traverse $\psi_b$ bottom-up and foreach node $\eta$ in $\psi_b$ do
    if $\eta$ is unit and $\eta$ has more than one child then
        add $\eta$ to unitsQueue;
        remove $\eta$ from $\psi_b$;
    end
end
return (unitsQueue, $\psi_b$);
```

Algorithm 2: `CollectUnits`

in the clause of η. Difficulties with such dependencies can be easily avoided if we reinsert the upper unit node η' after reinserting the unit node η (i.e. after reinsertion, η' must appear below η, to cancel the extra literal $\overline{\ell}$ from η's clause). This can be ensured by collecting the unit nodes in a queue during a bottom-up traversal of the proof and reinserting them in the order they were queued.

The algorithm for fixing a proof containing many roots performs a top-down traversal of the proof, recomputing the resolvents and replacing broken nodes (e.g. nodes having `deletedNodeMarker` as one of their parents) by their surviving parents (e.g. the other parent, in case one parent was `deletedNodeMarker`).

When unit nodes are collected and removed from a proof of a clause κ and the proof is fixed, the clause κ' in the root node of the new proof is not equal to κ anymore, but contains (some of) the duals of the literals of the unit clauses that have been removed from the proof. The reinsertion of unit nodes at the bottom of the proof resolves κ' with the clauses of (some of) the collected unit nodes, in order to obtain a proof of κ again.

```
input  : A proof $\psi_f$ (with a single root) and a queue q of root nodes
output: A proof $\psi'$
$\psi' \leftarrow \psi_f$;
while q $\neq \emptyset$ do
    $\eta \leftarrow$ first element of q;
    q $\leftarrow$ tail of q;
    if $\eta$ is resolvable with root of $\psi'$ then
        $\psi' \leftarrow$ resolvent of $\eta$ with the root of $\psi'$ ;
    end
end
return $\psi'$;
```

Algorithm 3: `ReinsertUnits`

Example 9. When applied to the proof ψ shown in Example 2, the algorithm `LU` collects the nodes η_4 and η_1, and replaces them by `deletedNodeMarker` (`DNM`):

$$\dfrac{\dfrac{\texttt{DNM} \qquad \dfrac{\eta_2 : a, c, \neg b \qquad \dfrac{\texttt{DNM} \qquad \eta_3 : a, b}{\eta_4 : \texttt{DNM}}\,a}{\eta_5 : a, c}\,b}{\eta_6 : c}\,a \qquad \dfrac{\dfrac{\texttt{DNM} \qquad \eta_7 : a, \neg b, \neg c}{\eta_8 : a, \neg c}\,b \qquad \texttt{DNM}}{\eta_9 : \neg c}\,a}{\psi : \bot}\,c$$

Fixing removes the `DNM`s. The derived unit clause η_4 is replaced by η_3, since its other parent was a `DNM`. And the proof ψ becomes:

$$\dfrac{\eta_2 : a, c, \neg b \qquad \eta_7 : a, \neg b, \neg c}{\psi : a, \neg b}\,c$$

Finally, the collected units η_4 (now replaced by η_3) and η_1 can be reinserted in the bottom of the proof, resulting in $((\eta_2 \odot \eta_7) \odot \eta_3) \odot \eta_1$:

$$\dfrac{\dfrac{\dfrac{\eta_2 : a, c, \neg b \qquad \eta_7 : a, \neg b, \neg c}{\psi : a, \neg b}\,c \qquad \eta_3(\eta_4) : a, b}{\psi' : a}\,b \qquad \eta_1 : \neg a}{\psi'' : \bot}\,a$$

□

For efficiency reasons, modern SAT solvers tend to use unit clauses eagerly: once a unit clause is found, it is used to simplify all other clauses. While this is clearly a good heuristic during proof search, unit resolutions can be delayed once a proof is found, since the number of resolution steps can then be significantly reduced. This effect is illustrated by a toy example in the proof of Theorem 10 below. While modern SAT solvers can produce a linear-size proof for this particular example, it nevertheless illustrates the undesirable effects that eager unit resolution may have on proof size.

Theorem 10. *There is a sequence of unsatisfiable clause sets S_n for which the shortest refutations φ_n obtained via eager unit resolution grow quadratically (i.e. $|\varphi_n| \in \Omega(n^2)$) while the compressed proofs $\texttt{LU}(\varphi_n)$ grow only linearly (i.e. $|\texttt{LU}(\varphi_n)| \in O(n)$).*

Proof. Consider the clause set S_n below:

$$\kappa_1 = \neg p_1 \quad \kappa_2 = p_1, \neg p_2 \quad \kappa_3 = p_1, p_2, \neg p_3 \quad \ldots \quad \kappa_{n+1} = p_1, p_2, p_3, \ldots, p_n$$

By *eager unit resolution*, κ_1 is firstly resolved with all other n clauses. Then the unit resolvent of κ_1 and κ_2 is resolved with all resolvents of κ_1 and κ_i ($3 \leq i \leq n+1$) and so on... The k^{th} iteration of unit resolution generates $n + 1 - k$ resolvents. One of these is the unit clause $\kappa^u_{k+1} = \neg p_{k+1}$ which is then resolved in the next iteration. It is easy to see that this refutation φ_n has length $\frac{n^2+n}{2}$. The compressed proof $\texttt{LU}(\varphi_n)$, shown below, has length equal to n only.

$$(\kappa_1 \odot (\ldots \odot (\kappa_{n-1} \odot (\kappa_n \odot \kappa_{n+1})) \ldots))$$

□

5 Algorithm `RecyclePivotsWithIntersection`

Our second algorithm, `RecyclePivotsWithIntersection` (`RPI`), aims at compressing irregular proofs. It can be seen as a simple but significant modification of the `RecyclePivots` (`RP`) algorithm described in [1], from which it derives its name. Although in the worst case full regularization can increase the proof length exponentially [10], 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 $\psi$
output: A possibly less-irregular proof $\psi'$
$\psi' \leftarrow \psi$;
traverse $\psi'$ bottom-up and foreach node $\eta$ in $\psi'$ do
    if $\eta$ is a resolvent node then
        setSafeLiterals($\eta$) ;
        regularizeIfPossible($\eta$)
    end
end
$\psi' \leftarrow$ fix($\psi'$) ;
return $\psi'$;
```

Algorithm 4: `RecyclePivotsWithIntersection`

```
input  : A node $\eta$
output: nothing (but the proof containing $\eta$ may be changed)
if $\eta$.rightResolvedLiteral $\in$ $\eta$.safeLiterals then
    replace left parent of $\eta$ by deletedNodeMarker ;
    mark $\eta$ as regularized
else if $\eta$.leftResolvedLiteral $\in$ $\eta$.safeLiterals then
    replace right parent of $\eta$ by deletedNodeMarker ;
    mark $\eta$ as regularized
end
```

Algorithm 5: `regularizeIfPossible`

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 5. 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 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

```
input  : A node $\eta$
output: nothing (but the node $\eta$ gets a set of safe literals)
if $\eta$ is a root node with no children then
    $\eta$.safeLiterals $\leftarrow$ $\eta$.clause
else
    foreach $\eta' \in \eta$.children do
        if $\eta'$ is marked as regularized then
            safeLiteralsFrom($\eta'$) $\leftarrow$ $\eta'$.safeLiterals ;
        else if $\eta$ is left parent of $\eta'$ then
            safeLiteralsFrom($\eta'$) $\leftarrow$ $\eta'$.safeLiterals $\cup$ { $\eta'$.rightResolvedLiteral } ;
        else if $\eta$ is right parent of $\eta'$ then
            safeLiteralsFrom($\eta'$) $\leftarrow$ $\eta'$.safeLiterals $\cup$ { $\eta'$.leftResolvedLiteral } ;
        end
    end
    $\eta$.safeLiterals $\leftarrow \bigcap_{\eta' \in \eta.\text{children}}$ safeLiteralsFrom($\eta'$)
end
```

Algorithm 6: `setSafeLiterals`

```
input  : A node $\eta$
output: nothing (but the node $\eta$ gets a set of safe literals)
if $\eta$ is a root node with no children then
    $\eta$.safeLiterals $\leftarrow \emptyset$
else
    if $\eta$ has only one child $\eta'$ then
        if $\eta'$ is marked as regularized then
            $\eta$.safeLiterals $\leftarrow \eta'$.safeLiterals ;
        else if $\eta$ is left parent of $\eta'$ then
            $\eta$.safeLiterals $\leftarrow \eta'$.safeLiterals $\cup$ { $\eta'$.rightResolvedLiteral } ;
        else if $\eta$ is right parent of $\eta'$ then
            $\eta$.safeLiterals $\leftarrow \eta'$.safeLiterals $\cup$ { $\eta'$.leftResolvedLiteral } ;
        end
    else
        $\eta$.safeLiterals $\leftarrow \emptyset$
    end
end
```

Algorithm 7: `setSafeLiterals` for `RecyclePivots`

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.

Example 11. When applied to the proof ψ shown in Example 2, the algorithm `RPI` assigns $\{a, c\}$ and $\{a, \neg c\}$ as the safe literals of, respectively, η_5 and η_8. The safe literals of η_4 w.r.t. its children η_5 and η_8 are respectively $\{a, c, b\}$ and $\{a, \neg c, b\}$, and hence the safe literals of η_4 are $\{a, b\}$ (the intersection of $\{a, c, b\}$

and $\{a, \neg c, b\}$). Since the right resolved literal of η_4 (a) belongs to η_4's safe literals, η_4 is correctly detected as a redundant node and hence regularized: η_4 is replaced by its right parent η_3. The resulting proof is shown below:

$$\dfrac{\dfrac{\eta_1 : \neg a \qquad \dfrac{\eta_2 : a, c, \neg b \qquad \eta_3 : a, b}{\eta_5 : a, c}}{\eta_6 : c} \qquad \dfrac{\dfrac{\eta_3 \qquad \eta_7 : a, \neg c, \neg b}{\eta_8 : a, \neg c} \qquad \eta_1}{\eta_9 : \neg c}}{\psi : \bot}$$

$$(\underbrace{\{\neg a\}}_{\eta_1} \odot (\{a, c, \neg b\} \odot \underbrace{\{a, b\}}_{\eta_3})) \odot ((\eta_3 \odot \{\neg b, \neg c, a\}) \odot \eta_1)$$

RP, on the other hand, assigns $\emptyset$ as the set of safe literals for η_4. Therefore, it does not detect that η_4 is a redundant irregular node, and then $\mathtt{RP}(\varphi) = \varphi$. □

Theorem 12. *For any proof φ, $|\mathtt{RPI}(\varphi)| \leq |\mathtt{RP}(\varphi)|$.*

Proof. For every node η in φ, let $S^{\eta}_{\mathtt{RPI}}$ (resp., $S^{\eta}_{\mathtt{RP}}$) be the set of safe literals for η computed by RPI and RP. It is easy to see that $S^{\eta}_{\mathtt{RPI}} \supseteq S^{\eta}_{\mathtt{RP}}$ for all η. Therefore, RPI detects and eliminates more redundancies than RP. □

The better compression of RPI does not come for free, as computing an intersection of sets is more costly than assigning the empty set. For a node η with k children, k sets must be intersected and the size of each set is in the worst case in $O(h)$, where h is the length of the shortest path from η to a root.

6 Experimental Evaluation

In order to evaluate these algorithms, we implemented prototypes[1] of RP, RPI, and LU in the high-level programming language Scala [8] and applied them, as well as the two possible sequential compositions of LU and RPI, to 98 refutations of standard unsatisfiable benchmark problems[2]. These refutations[3] were generated by the CDCL-based SAT-solver included in veriT [4]. For each proof ψ and each algorithm α, we measured[4] the time $t(\alpha, \psi)$ taken by α to compress ψ and the lengths of ψ and $\alpha(\psi)$, and we calculated the obtained compression $((|\psi| - |\alpha(\psi)|)/|\psi|)$ and the compression speed $((|\psi| - |\alpha(\psi)|)/t(\alpha, \psi))$.

The scatter plot shown in the left side of Figure 1 confirms that RPI always compresses more than RP, as predicted by Theorem 12. Furthermore, it shows that RPI often compressed much more than RP. The comparison becomes even more favorable when RPI is followed by LU, as shown in the right-hand figure.

[1] Source code available at `http://code.google.com/p/proof-compression/`.

[2] The benchmarks were: (1) unsatisfiable problems of the SatRace 2010 competition solved by veriT in less than 30s and stemming from verification of software ("Babic" and "Nec" benchmarks) and hardware ("IBM" and "Manolios"); (2) smaller problems of the Sat-Lib DIMACS benchmarks ("AIM", "Dubois", "JNH", "BF", "Pret", "SSA", described in `www.cs.ubc.ca/~hoos/SATLIB/benchm.html`)

[3] Proofs in `www.logic.at/people/bruno/Experiments/2011/LU-RPI/Proofs.zip`

[4] The raw data of the experiments is available at `https://spreadsheets.google.com/ccc?key=0Al709ihGgKdndG1yWm5kNXIzNHppNXd0ZGQwTE01V0E&hl=en`.

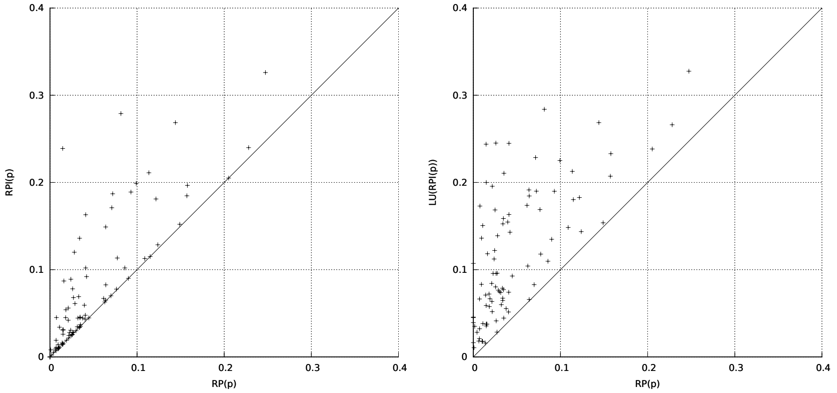


Fig. 1: Comparing RP and RPI, resp. RPI followed by LU.

Even though our implementations are just prototypes and the experiments were executed on a modest computer (2.8GHz Intel Core 2 Duo processor with only 2GB of RAM (1067MHz DDR3) available for the Java Virtual Machine), we were pleased to see that the algorithms presented an acceptable and scalable performance. The proofs generated by veriT contained up to millions of derived clauses and were up to 100MB big (in the size of the text file) in a minimalistic proof format[5]. They included all intermediate clauses that had been learned during the execution of veriT, even those that were not used to derive the final empty clause. Before applying the compression algorithms, we removed these unused clauses, but the pruned proofs were still up to more than half a million clauses long and up to about 20MB big. The execution times of all algorithms varied between less than 100 miliseconds for the smaller proofs, less than 10 seconds for the majority of the much bigger proofs of the SatRace benchmarks, and 7.5 minutes in the worst case (for a highly redundant proof with more than half a million clauses).

Figure 2 shows the compression (top) and compression speed (bottom) for the examples from the SatRace. The top figure suggests a trend where longer proofs are more redundant and allow for more compression. This might be due to the fact that the SAT solver backtracks and restarts more often for harder problems that generate longer proofs.

The bottom figure shows that compression speeds of the RPI and RP algorithms are very similar, although RPI took significantly more time than RP for some examples. In cases where the compression rates are comparable, the execution times are similar as well. When RPI took more time than RP, it achieved correspondingly better compression. This indicates that computing the inter-

[5] This format is closely related to the resolution proof terms used in this paper and is quite compact: (1) only the parents of a derived clause must be indicated explicitly; (2) a clause only needs an explicit name/number if it has more than one child.

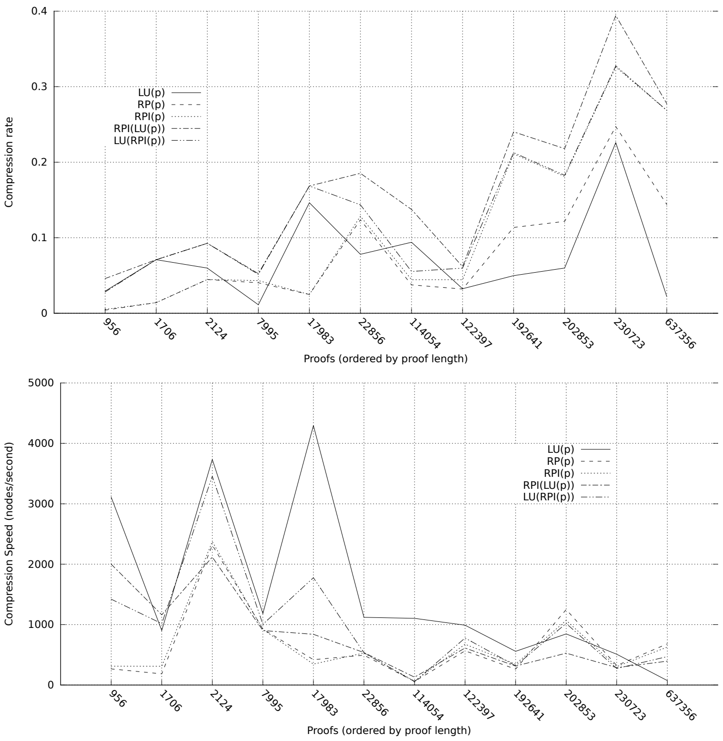
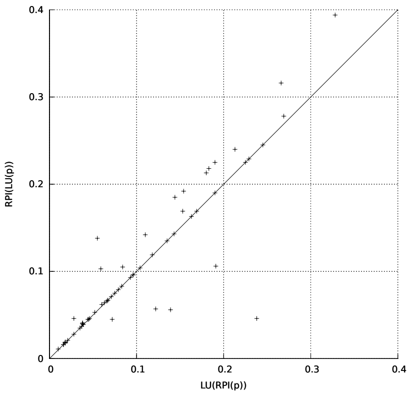


Fig. 2: Compression and compression speed for the SatRace examples.

sections is worthwhile in practice. Finally, note that LU is usually the fastest algorithm in terms of compression speed.



The compression achieved by applying both LU and RPI is usually less than the sum of the compressions achieved by each algorithm alone. This is so because certain redundancies are eliminated by both algorithms. Moreover, the scatter plot above shows that the order in which LU and RPI are applied matters.

7 Related Work and Ideas for Future Work

One of the kinds of local redundancy was considered in our previous work [6], where we also proposed *resolution hypergraphs* as a possible non-linear notation for resolution proofs, making it easier to identify and address non-local redundancies. Although in principle more general than the techniques described here, they do not scale to large proofs, because resolution hypergraphs can be exponentially larger than the proofs they represent.

The same kind of local redundancy was also mentioned by Simone et al. [9], as the local proof rewriting rule $A1'$. They address global redundancies by having another local proof rewriting rule ($A2$) that performs inference permutations when possible. As we have argued before, this approach is inherently inefficient, since too many permutations would have to be considered in order to eliminate all global redundancies. They also consider other interesting local proof rewriting rules that eliminate redundancies not considered in this paper. It would be worthwhile to generalize these other kinds of local redundancy by defining their global counterparts too; it might then be possible to adapt the global techniques described in this paper to these other kinds of redundancy.

Besides RP, Bar-Ilan et al. [1] also defined the RecycleUnits algorithm, which replaces one of the parents of a resolvent with a resolved literal ℓ by a unit clause containing ℓ, if such a unit clause exists somewhere else in the proof. Although this algorithm eliminates some kinds of redundancy, it generates redundancies of the kind handled by LU. Therefore it would be helpful to always execute LU after RecycleUnits, or to combine both algorithms more tightly: instead of replacing a parent by the unit, the resolvent can be replaced by the other parent, and the unit can be queued to be reinserted at the bottom of the proof.

Cotton [5] proposes to split a refutation ψ into a proof ψ_p of the unit clause containing the atom p and a proof $\psi_{\neg p}$ of unit clause containing the literal $\neg p$. This is done by deleting one of the parents of every resolvent with pivot p. A new refutation ψ', possibly shorter than ψ, is then obtained by resolving ψ_p and $\psi_{\neg p}$. Since in ψ' there is now only one resolvent with pivot p, all potential redundancies with pivot p are removed with this splitting technique. Consequently, in principle this splitting technique could subsume all other techniques previously described, including the ones in this paper. However, since not all potential redundancies are actual redundancies, ψ' might actually be longer than ψ. This problem is atenuated by heuristically choosing a promising literal p to split, and iterating until the next proof ψ' becomes longer than the current proof ψ. The techniques that globally identify precisely which potential redundancies are actual redundancies, such as those presented in [1] and here should scale better, since they do not need to iterate an undefined number of times and fix the proof after every iteration.

While this paper focused on regularization of proofs, trying to compress proofs by introducing irregularities is also an interesting possibility to be investigated in future work, since exponential compression might be achieved in the best cases.

8 Conclusions

The use of proof contexts makes for a clear transition from local to global transformations of proofs, and in particular helped us generalize certain kinds of local redundancies to global ones. In this way, we designed two algorithms that eliminate these global redundancies more efficiently than previous ones. Our experiments seem to indicate that we can expect reductions of around 20% for large proofs, beyond what is possible just by pruning irrelevant inferences. Since these reductions essentially come for free and proof checking can be costly (for example when it is performed by the trusted kernel of an interactive proof assistant or when it has to be repeated many times by different proof consumers such as in a PCC scenario), we believe that it is worthwhile to implement our techniques when proof size matters.

References

1. O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-time reductions of resolution proofs. *Hardware and Software: Verification and Testing*, pages 114–128, 2009.
2. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.
3. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
4. T. Bouton, D. Caminha B. de Oliveira, D. Deharbe, and P. Fontaine. verit: an open, trustable and efficient smt-solver. In R. A. Schmidt, editor, *Automated Deduction - CADE-22 (22nd International Conference on Automated Deduction)*, 2009.
5. S. Cotton. Two techniques for minimizing resolution proofs. In *13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, 2010.
6. P. Fontaine, S. Merz, and B. Woltzenlogel Paleo. Exploring and Exploiting Algebraic and Graphical Properties of Resolution. In *8th International Workshop on Satisfiability Modulo Theories (Part of FLoC - Federated Logic Conferences)*, 2010.
7. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.
8. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala.* Artima Press, 2008.
9. S. Simone, R. Brutomesso, and N. Sharygina. An efficient and flexible approach to resolution proof reduction. In *6th Haifa Verification Conference*, 2010.
10. G. Tseitin. On the complexity of proofs in propositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*, volume 2. Springer-Verlag, 1983.