# The Skeptik Proof Compression System

Joseph Boudou[1] *, Andreas Fellner[2] **, and Bruno Woltzenlogel Paleo[3] ***

[1] IRIT, Université de Toulouse, France
joseph.boudou@irit.fr
[2] Free University of Bolzano, Italy
fellner.a@gmail.com
[3] Vienna University of Technology, Austria
bruno@logic.at

**Abstract.** This paper describes Skeptik: a system for compressing and improving proofs obtained by automated reasoning tools.

## 1 Introduction

## 2 User Interface

## 3 Supported Proof Formats

### 3.1 TraceCheck Format

### 3.2 The veriT Proof Format

### 3.3 The Skeptik Proof Format

## 4 Available Proof Compression Algorithms

### 4.1 RPI

### 4.2 LU

### 4.3 LowerUnivalents

The `LowerUnivalents` algorithm [**?**] (`LUniv`) generalizes `LowerUnits` by exploiting the information of the already lowered subproofs and their pivots. Then, it becomes possible to lower a non-unit node if all its conclusion's literals but one can be deleted by the already lowered subproofs. Moreover, some partial regularization based on the lowered pivots can be achieved simultaneously.

In Skeptik, `LUniv` has been implemented as a replacement for the `fixProof` function used by some algorithms to reconstruct a proof after deletions. Therefore, non-sequential combinations of those algorithms with `LUniv` are easy to implement. For instance, `LUnivRPI` is a non-sequential combination of `LUniv` after `RPI`. This latter algorithm is currently one the best trade-off between compression time and compression ratio [**?**].

* Supported by the Google Summer of Code 2012 program.
** Supported by the Google Summer of Code 2013 program.
*** Supported by the Austrian Science Fund, project P24300.

### 4.4 RPI[3]LU and RPI[3]LUniv

### 4.5 ReduceAndReconstruct

The `ReduceAndReconstruct` (`RedRec`) approach consists in applying local transformation rules to each node. The set of local rules presented in [4] are sufficient to emulate any other compression algorithm. Unfortunately, the proposed heuristic only consists in trying to apply each rule (with given priorities) to each node in a top-down traversal. For this heuristic to be efficient, the process has to be repeated many times. The resulting algorithm can achieve very good compression ratio if run long enough.

As this algorithm allows experimentations in the local transformation rules, the heuristic to apply them and the termination conditions, its implementation in Skeptik is very modular. Each component is defined independently and a convenient framework allows to combine them as desired. A handful of alternative local transformation rules and termination conditions have been implemented too.

### 4.6 Split

The Split [2] algorithm is a technique to lower pivot variables in a proof.
From a proof with conclusion $C$, two proofs with conclusion $v \vee C$ and $\neg v \vee C$ are constructed, where the variable $v$ is chosen heuristically.
In a first step the positive/negative premises of resolvents with pivot $v$ are removed from the proof. Afterwads the proof is fixed, by traversing it top-down and fixing each proof node. A proof node is fixed by either replacing it by one of or resolving the fixed premises.
The roots of the resulting proofs are resolved, using $v$ as pivot, to obtain a new proof of $C$.
The time-complexity of this algorithm is linear in the proof length and it suits very well for repeated application. This can be done iteratively or recursively. Also multiple variables can be chosen in advance. All these variants are implemented into Skeptik.

### 4.7 TautologyElimination

### 4.8 StructuralHashing

### 4.9 DAGification

### 4.10 Subsumption

Subsumption based algorithms use, as the name implies, the subsumption relation on clauses for compressing a proof. A clause $C_1$ subsumes a clause $C_2$ *iff* every literal occuring in $C_1$ also occurs in $C_2$. The general goal is to replace a clause $C$ by another clause $D$, used elsewhere in the proof, that are such that $C$ subsumes $D$. There are three subsumption based compression algorithms implemented into Skeptik.

**Top-down Subsumption** searches for subsumed clauses among all clauses visited earlier in a top-down traversal. The time-complexity of this algorithm is worst case quadratic in the number of proof nodes.
**Bottom-up Subsumption** searches for subsumed clauses among all clauses visited earlier in a bottom-up traversal. A subsumed clause $D$ can only replace a clause $C$, if $D$ is not an ancestor of $C$ in the graph representing the proof. Bottom-up Subsumption has the same time-complexity as Top-down Subsumption, but the additional ancestor-check makes it slower in practise.
**RecycleUnits** [1] is a special case of Bottom-up Subsumption that only searches for subsumed clauses that are unit (i.e. contain only one literal). This algorithm has a worst case time-complexity that is quadratic in the number of unit clauses.

### 4.11 Pebbling

Pebbling algorithms compress proofs in the **space measure**, as opposed to the usual length compression. The space measure of a proof indicates how many proof nodes maximally have to be kept in memory, while reading the proof, simultaniously.
This measure is closely related to the **Black Pebbling Game** [3], which lends its name to the algorithms described here. A strategy for this game directly corresponds to a topological node ordering, i.e. an ordering of nodes such that every premise of each node is lower than the node itself, combined with deletion information, i.e. extra lines in the proof output indicating that a node can be deleted from memory. An optimal strategy for the Black Pebbling Game would therefore result in optimal space compression of the proof. However, as shown in [3], finding the optimal solution is PSPACE-complete and therefore not a feasable approach for this program.
To obtain algorithms that have an acceptable runtime greedy heuristics for finding a good node ordering are used.
**Top-down pebbling** directly corresponds to playing the game with limited information on how the proof looks. At every point there are nodes which can be pebbled (initially these are only the axioms). Using information from these nodes and their children nodes, it is decided which node to pebble next, which can make other nodes pebblable. This is done until the root node is pebbled. Unfortunately the lack of knowledge about the structure of the proof often results in bad space compression.
**Bottom-up pebbling** constructs a node ordering by visiting proof nodes and their premises recursively, starting from the root node. At every node it is chosen heuristically in what order its premises are visited. After all premises are visited, the node is added to the oder.

## 5 Implementation Details

ToDo: statistics about the code

### 5.1 Organization of the Code

ToDo: brief description of the package structure

### 5.2 Data Structures for Formulas

### 5.3 Data Structures for Proofs

## 6 Conclusions and Future Work

ToDo: limitation: memory consumption in Scala underlying symbols are strings

## References


1. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Reducing the size of resolution proofs in linear time. STTT 13(3), 263–272 (2011)
2. Cotton, S.: Two techniques for minimizing resolution proofs. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing SAT 2010, Lecture Notes in Computer Science, vol. 6175, pp. 306–312. Springer (2010)
3. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. SIAM Journal on Computing 9(3), 513–524 (1980)
4. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, vol. 6504, pp. 182–196. Springer (2011)