# Skeptik [System Description]

Joseph Boudou[1], Andreas Fellner[2,3], 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 introduces Skeptik: a system for checking, compressing and improving proofs obtained by automated reasoning tools.

## 1 Introduction

There are various reasons why it is desirable for automated reasoning tools to output not only a *yes* or *no* answer to a problem but also *proofs/refutations* or *(counter)models*. Firstly, state-of-the-art tools are complex and heavily optimized. Their code is often long, hard to understand and difficult to automatically verify. Consequently, the *yes/no* answers cannot be fully trusted, unless they are accompanied by proofs or (counter)models that serve as independently checkable certificates of their correctness.

Furthermore, for most applications, a *yes/no* answer is inherently insufficient. We often already known in advance whether a problem is expected to be satisfiable or unsatisfiable, and we want more than just a confirmation of this expectation. For satisfiable formulas, the desired information is encoded in the model; while for valid formulas, it is contained in the proof. In case the expectation was wrong, refutations and countermodels can be very helpful to explain issues in the encoding of the problem, in order to correct and refine it.

Although current automated deduction tools are very efficient at finding proofs, they do not necessarily find the best proofs. Skeptik efficiently finds and eliminates redundancies in proofs, thus compressing and improving them according to various metrics (http://github.com/Paradoxika/Skeptik/).

*Related Work:* CERes (http://www.logic.at/ceres) [?] is another proof transformation system, especialized in cut-elimination for classical first-order sequent calculus. It was replaced by GAPT (http://code.google.com/p/gapt/) [?], extended with cut-introduction techniques [?]. MINLOG (http://www.mathematik.uni-muenchen.de/ logik/minlog/) [?] extracts functional programs from proofs, employing a refined A-translation for functional proofs.

* Funded by Google Summer of Code 2012 and 2013 and FWF project P24300.

## 2 Installation

Skeptik is implemented in Scala and runs on the java virtual machine (JVM). Therefore, Java (https://www.java.com/) must be installed. The easiest way to download Skeptik is via `git` (http://git-scm.com/), by executing [`git clone git@github.com:Paradoxika/Skeptik.git`] in the folder where Skeptik should be downloaded. It is helpful to install `SBT` (http://www.scala-sbt.org/), a build tool that automatically downloads all compilers and libraries on which Skeptik depends. To compile, build and package Skeptik, run [`sbt one-jar`] in Skeptik's home folder. This generates a jar file that can be executed like any other Java jar file.

`SBT` and Scala programs may need a lot of memory for compilation and execution. If out-of-memory problems occur, the JVM's maximum available memory can be increased by executing [`export JAVA_TOOL_OPTIONS=''-Xmx1024m''`] in the terminal.

## 3 Usage

The command [`java -jar skeptik.jar --help`] displays a help message explaining how to use Skeptik. For example, to compress the proof "eq_diamond9.smt2" using the algorithm `RPI` and write the compressed proof using the 'smt2' proof format, the following command should be executed: [`java -jar skeptik.jar -a RPI -f smt2 examples/proofs/VeriT/eq_diamond9.smt2`]. Skeptik can be called with an arbitrary number of algorithms and proofs. The following command would compress the proofs "p1.smt2" and "p2.smt2" with two algorithms each (`RP` and a sequential composition of `DAGify`, `RPI` and `LU`):
[`java -jar skeptik.jar -a RP -a (DAGify*RPI*LU) p1.smt2 p2.smt2`]

## 4 Implementation Details

In Skeptik every logical expression is a simply typed lambda expression, implemented by the abstract class `E` with concrete subclasses `Var`, `App` and `Abs` for, respectively, *variables*, *applications* and *abstractions*. Scala's *case classes* are used to make `E` behave like an algebraic datatype with (pattern-matchable) constructors `Var`, `App` and `Abs` à la functional programming.

Skeptik is flexible w.r.t. the underlying proof calculus. Every proof node is an instance of the abstract class `ProofNode` and must contain a judgment of some subclass `J` of `Judgment` and a (possibly empty) collection of premises (which are other proof nodes). A proof is a directed acyclic graph of proof nodes; it is implemented as the class `Proof`, which provides higher-order methods for traversing proofs. Thanks to Scala's syntax conventions, these methods can be used as an internal domain specific language that integrates harmoniously with the scala language itself.

A proof calculus is a collection of inference rules, implemented as concrete subclasses of `ProofNode`. In particular, the main inference rules of the propositional resolution calculus used for representing proofs generated by sat- and smt-solvers are `Axiom` and `R` (for resolution). They use a `Sequent` class (a subclass of `Judgment`) to represent clauses.

Classes for expressions and proof nodes are small and correctness conditions are checked during object construction. Once constructed, they cannot be changed, because they are *immutable*. Therefore, incorrect expressions and proofs cannot result from the transformations performed by Skeptik. Auxiliary functionality is not implemented in the classes but in their homonymous *companion objects*. Therefore, even though Skeptik has more than 21000 lines of code, its most critical core data structures are less than a few hundred lines long.

## 5 Supported Proof Formats

Scala's combinator parsing library makes it easy to implement parsers for various proof formats. Skeptik uses the extension of a file to determine its proof format. To export proofs in various formats, Skeptik provides many classes in exporters package that extend Java's `java.io.Writer` class.

The currently available proof formats are:

*The SMT-Lib Proof Format:* veriT
ToDo (Bruno)

*TraceCheck Format:* ToDo (Andreas)
ToDo: RUP
cite Armin Biere's sat-solver

Skeptik*'s Proof Format:* ToDo (Joseph)
ToDo: with deletion

## 6 Proof Compression Algorithms

Most algorithms for the compression of propositional resolution proofs generated by sat-solvers and smt-solvers described in the literature are available in Skeptik. They are shortly described below:

*RecyclePivots* (`RP`) [1, 2] compresses a proof by partially regularizing it. A proof is *irregular* [9] if the resolved literal of a resolution proof node is resolved again on the path from this node to the root node. `RP` finds irregular nodes efficiently by traversing the proof from the root to the leaves a single time and memorizing which literals were resolved. When it finds an irregular node, it marks one of its premises for deletion. In a second traversal, from the leaves to the root, irregular nodes are replaced by their non-deleted premises. As full regularization can lead to an exponential blow-up in the proof length [7], it is important to regularize

carefully and only partially. `RP` achieves this by resetting the set of literals for a node to the empty set when it has more than one child (i.e. when it is the premise of more than one node).

*RecyclePivotsWithIntersection* (`RPI`) [5] differs from `RP` in the treatment of a node with more than one child. Instead of resetting its set of literals to the empty set, the intersection of the sets of literals incoming from its children is computed. In this manner, the exponential blow-up is still avoided, but strictly more irregular nodes are detected and regularized.

### 6.1 LowerUnits

Another kind of proof redundancy happens when a node $\eta$ appears as premise of many resolutions on the same pivot $p$. In that case, it would be better to resolve $\eta$ on $p$ only once. The `LowerUnits` (`LU`) algorithm [5] reduces this kind of redundancy by lowering *units* down to the root of the proof. Units are subproofs whose conclusion consists in a single literal. Such a subproof can always be lowered down the proof.

`LowerUnits` is of linear time complexity in the length of the proof. It is a very fast algorithm which perform good compression ratio.

### 6.2 LowerUnivalents

The `LowerUnivalents` algorithm [3] (`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, the lowered pivots are safe literals and thus some partial regularization 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 [3].

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

`RPI[3]LU` and `RPI[3]LUniv` are non-sequential combinations of `RPI` after `LU` and `LUniv`. They consist in three traversals. The first traversal collects subproofs to be lowered down the proof. The second traversal computes the sets of safe literals for each node, taking into account the subproofs marked for being lowered. The last traversal actualy compress the proof by removing redundant branches and lowering subproofs.

These algorithm are optimizations of the corresponding sequential compositions, achieving the same compression ratio in less time.

### 6.4 ReduceAndReconstruct

The `ReduceAndReconstruct` (`RedRec`) approach consists in applying local transformation rules to each node. The set of local rules presented in [8] 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.

### 6.5 Split

The Split [4] 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.

### 6.6 TautologyElimination

### 6.7 StructuralHashing

### 6.8 DAGification

### 6.9 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** [2] 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.

### 6.10 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, simultaneously.
This measure is closely related to the **Black Pebbling Game** [6], 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 [6], 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.

## 7 Conclusions and Future Work

ToDo: limitation: memory consumption in Scala underlying symbols are strings
New proof formats on demand

extension to first-order
contextual natural deduction
cut-introduction
Vienna Scientific Cluster

## References


1. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: Chockler, H., Hu, A.J. (eds.) Haifa Verification Conference. Lecture Notes in Computer Science, vol. 5394, pp. 114–128. Springer (2008)
2. 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)
3. Boudou, J., Paleo, B.W.: Compression of propositional resolution proofs by lowering subproofs. In: Galmiche, D., Larchey-Wendling, D. (eds.) TABLEAUX. Lecture Notes in Computer Science, vol. 8123, pp. 59–73. Springer (2013)
4. 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)
5. Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Compression of propositional resolution proofs via partial regularization. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE. Lecture Notes in Computer Science, vol. 6803, pp. 237–251. Springer (2011)
6. 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)
7. Goerdt, A.: Comparing the complexity of regular and unrestricted resolution. In: Marburger, H. (ed.) GWAI. Informatik-Fachberichte, vol. 251, pp. 181–185. Springer (1990)
8. 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)
9. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning: Classical Papers in Computational Logic 1967-1970, vol. 2. Springer-Verlag (1983)