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, especially sat- and smt-solvers.

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.

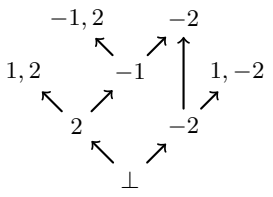
Classes for expressions and proof nodes are small and correctness conditions are checked during object construction. Once constructed, they cannot be changed, because they are *immutable*. Therefore, incorrect expressions and proofs cannot result from the transformations performed by Skeptik. Auxiliary 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. Available proof formats are:

TraceCheck Format: The TraceCheck [**?**] format is one of the three formats accepted at the *Certified Unsat* track of the SAT-Competition and is used by sat-solvers such as `PicoSAT` [**?**]. Each line declares a new clause, specifying its name (a fresh positive integer), a space separated list of literals (positive or negative integers, depending on the polarity of the literals), and a list of premises (other clauses, referred to by their names) needed to derive the new clause by regular input resolution. Zero is used as a delimiter. A detailed description can be found at http://fmv.jku.at/tracecheck/. An example is shown in Figure 1.

Other formats accepted at the *Certified Unsat* track are less detailed and hence less convenient to be used by tools that post-process proofs. The omission of premises in the RUP format, for example, throws away information about the DAG structure of the proofs. Nevertheless, there are tools for converting RUP proofs to a resolution format that resembles the trace-check format [**?**].




```
1 1  2 0 0
2 -1  2 0 0
3 -2  0  0
4 1  -2 0 0
5 2  0 1 2 3 0
6 -2  0  3 4 0
7 0  5  6
```

Fig. 1: A proof and its representation in the TraceCheck format

The SMT-Lib Proof Format: veriT
ToDo

Skeptik*'s Proof Format:* ToDo
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`) [**?**,**?**] compresses a proof by partially regularizing it. A proof is *irregular* [**?**] if the resolved literal (pivot) 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 [**?**], 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`) [**?**] 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.

LowerUnits (`LU`) [**?**] partially eliminates a kind of redundancy that is almost orthogonal to irregularity. When a node η appears as premise of many resolutions with the same pivot p, it is desirable to resolve η on p only once instead. This is not always possible, unless η contains a unit clause (a clause with only the literal p). `LowerUnits` is able to reduce the redundancy by lowering all unit nodes. The nodes are removed from their places and reintroduced in the very bottom of the proof, by a resolving them (at most once) with the root of the fixed proof.

LowerUnivalents (`LUniv`) [**?**] generalizes `LowerUnits`. By keeping track of nodes that have already been lowered and their pivots, it becomes possible to lower a non-unit node if it is *univalent*: all its literals but one (its so-called *valent* literal) can be resolved against the valent literals of the already lowered nodes.

LUnivRPI [**?**] is a non-sequential combination of `LUniv` after `RPI` and currently provides one of the best trade-offs between compression time and compression ratio. Non-sequential combinations with `LUniv` are easy to implement, because `LUniv` has been implemented as a replacement for the `fixProof` function used by some algorithms to reconstruct a proof after deletions.

RPI[3]LU and RPI[3]LUniv are non-sequential combinations of `RPI` after `LU` and `LUniv`, respectively. They consist of three traversals. The first traversal collects subproofs to be lowered. The second traversal computes the sets of safe literals for each node, taking into account the subproofs marked for being lowered. The last traversal actually compresses 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.

Reduce&Reconstruct (`RR`) [**?**] applies local transformation rules that either eliminate local redundancies or shuffle the order of resolution steps (similarly to what Gentzen's rank reduction rules for cut-elimination do) in order to gradually transform non-local redundancies into local ones. Although the set of local rules used are sufficient to emulate any other compression algorithm, the algorithm may need many traversals to shuffle the order of resolutions steps sufficiently well to eliminate non-local redundancies. This algorithm can achieve very good compression ratio, if run for a long enough time. The implementation in Skeptik is very modular, allowing convenient experimentation with various alternative local transformation rules, heuristics to apply them and termination conditions. There still seems to be plenty of space to improve heuristics and termination conditions for this algorithm.

Split [**?**] lowers pivot variables in a proof. From a proof with conclusion C, two proofs with conclusions $v \vee C$ 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. Afterwards 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 its fixed premises or resolving them. 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, but it has to be repeated many times to obtain significant compression. This can be done iteratively or recursively. Also multiple variables can be chosen in advance. All these variants are implemented in Skeptik.

Tautology Elimination (`ET`) eliminates proof nodes containing tautological clauses. Although tautological clauses normally do not occur in proofs generated by sat- and smt-solvers, they may occur in proofs compressed by some of the algorithms described here.

DAGification (`D`) finds proof nodes having equal clauses and replaces each of them by only one of them.

Subsumption algorithms generalize DAGification by using the subsumption relation on clauses instead of the equality relation. A clause C_1 subsumes a clause C_2 iff every literal occurring in C_1 also occurs in C_2. The goal is to replace a node containing a clause C_2 by another node containing a clause C_1 if C_1 subsumes C_2. There are three subsumption-based proof compression algorithms implemented in Skeptik. `TopDownSubsumption` (`TDS`) 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. `BottomUpSubsumption` (`BUS`) 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. `BUS` has the same time-complexity as `TDS`, but the additional ancestor-check makes it much slower in practice. `RecycleUnits` (`RU`) [**?**] is a special case of `BUS` that only searches for subsuming clauses that are unit (i.e. contain only one literal). Its worst case time-complexity is quadratic in the number of unit clauses.

Pebbling algorithms [**?**] compress proofs w.r.t. their *space*, not their length. The space of a proof is the maximum number of proof nodes that have to be kept in memory simultaneously, while reading and checking the proof.
This measure is closely related to the **Black Pebbling Game** [**?**], which lends its name to the algorithms described here. A strategy for this game is a topological node ordering (i.e. an ordering of nodes such that every premise of each node is lower than the node itself) enriched 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 [**?**], deciding whether a proof can be pebbled using at most n pebbles is PSPACE-complete, so constructing the optimal solution is not a feasable approach for Skeptik. Greedy heuristics are used for finding a good, though not optimal, node ordering. `TopDownPebbler` (`TDP`) plays the game in a top-down manner. At every point there are nodes which can be pebbled. Initially only the axioms can be pebbled. Using information from the pebblable nodes and their children, heuristics decide which node to pebble next. If all premises of a node have been pebbled, it becomes pebblable. This is repeated until the root node is pebbled. Unfortunately, `TDP`'s heuristics have very limited knowledge about the local structure of the proof only and is often not able to achieve good compression. `BottomUpPebbler` (`BUP`) constructs a node ordering by visiting proof nodes and their premises recursively, starting from the root node. At every node, `BUP` heuristically chooses which subproof rooted in the premises of the node should be pebbled first. After all premises are pebbled, the node itself is pebbled. `BUP` is thus more aware of the global structure of the proof, and achieves much better space compression than `TDP`.

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