Extended Abstract: Combining a Logical Framework with an RUP Checker for SMT Proofs

Duckki Oe and Aaron Stump
Computer Science
The University of Iowa

Abstract

We describe work in progress on a new approach, and associated tools, for checking proofs produced by SMT solvers. The approach extends our previous work on LFSC ("Logical Framework with Side Conditions"), a meta-language in which different proof systems for different SMT solvers can be declaratively specified. In this paper, we show how the LFSC proof checker can delegate the checking of propositional inferences (within a proof of an SMT formula) to a propositional proof checker `clcheck` based on Reverse Unit Propagation (RUP). This approach shows promising improvements in proof size and proof checking time for benchmark proofs produced by the `clsat` QF_IDL solver. We also discuss work in progress to replace `clcheck` with a different RUP checker we are developing called `vercheck`, whose soundness we are in the process of statically verifying.

1 Introduction

The problem of devising a standardized proof format for SMT solvers is an ongoing challenge. A number of solvers are proof-producing; for example, CVC3, veriT, and Z3 all produce proofs, in different formats [1, 2, 5]. In previous work, we advocated for the use of a flexible meta-language for proof systems called LFSC ("Logical Framework with Side Conditions"), from which efficient proof-checkers could be generated by compilation [6, 7]. Our team at The University of Iowa is currently working on a new implementation of LFSC, intended for public release.

For many SMT problems, propositional reasoning is a large if not the dominating component of proofs. Compressing the size of propositional proofs is therefore of significant interest (see, e.g., [3]). In the current paper, we describe an approach, and tools in progress, to compress the size of such proofs, by using an external propositional proof checker called `clcheck`, based on the idea of Reverse Unit Propagation.

2 SMT Proofs in LFSC

In previous work, we have advocated the use of a meta-language called LFSC ("Logical Framework with Side Conditions") for describing proof systems for SMT solvers [6, 7]. We use a meta-language to avoid imposing a single proof system on all solvers. SMT solvers support many different *logics*, and different solving algorithms naturally give rise to different schemes for representing deductions. Pragmatically, it may not be realistic to ask solver implementors to support a specific axiomatization, which may not fit well with their internal data structures or algorithms. Instead, we are working towards a common meta-language, in which different proof systems may be described. This at least would establish a common meta-language

```
(declare var type)
(declare lit type)
(declare pos (! x var lit))
(declare neg (! x var lit))

(declare clause type)
(declare cln clause)
(declare clc (! x lit (! c clause clause)))
(declare concat (! c1 clause (! c2 clause clause)))
(declare in_and_remove (! l lit (! c clause clause)))
```

Figure 1: Data Structures in LFSC for Generalized Clauses

```
(declare holds (! c clause type))
(declare R (! c1 clause (! c2 clause
           (! u1 (holds c1)
           (! u2 (holds c2)
           (! n var
            (holds (concat (in_and_remove (pos n) c1)
                     (in_and_remove (neg n) c2)))))))))

(program simplify_clause ((c clause)) clause ...)

(declare satlem (! c1 clause
                (! c2 clause
                (! c3 clause
                (! u1 (holds c1)
                (! r (^ (simplify_clause c1) c2)
                (! u2 (! x (holds c2) (holds c3))
                   (holds c3)))))))))
```

Figure 2: LFSC Rules for Resolution Proofs

for comparison of proofs and for (meta-language) proof checkers, and could facilitate later adoption of at least a common core proof system for SMT. Other researchers are working towards similar goals, and we anticipate development of a common solution in the coming year [1].

Signatures. In LFSC, proof systems are described by *signatures*. Figures 1 and 2 give part of the signature we use to produce proofs from our `clsat` QF_IDL solver. Most of the 1000-line signature is elided here, including rules for CNF conversion and arithmetic reasoning. The rules shown were developed in our previous work [6], and defer binary resolutions (constructed using the `R` proof rule) until many of them can be processed at once when a lemma is added. Resolutions are deferred by constructing a *generalized clause* (the `clause` type declared in Figure 1) using the `concat` and `in_and_remove` constructors. These constructors represent deferred operations required in order to compute the actual binary resolvent. The side-condition program `simplify_clause` (code omitted from Figure 2) executes those deferred operations in an optimized way, to construct the final resolvent of a series of binary resolutions without constructing the intermediate resolvents.

Rules can be thought of as richly typed constructors, accepting arguments (via the `!` construct) whose

types may mention earlier arguments. For example, the `R` rule has 5 inputs: `c1`, `c2`, `u1`, `u2`, and `n`. The first two are mentioned in the types of the second two. As an optimization, arguments for `c1` and `c2` may be elided in proofs built using these proof rules, since their values can be determined during proof checking from the types of the arguments for `u1` and `u2`.

The rule `satlem` uses the caret (`^`) notation to invoke the `simplify_clause` side-condition program on a clause `c1`, to compute a clause `c2` without deferred operations `concat` and `in_and_remove`. The rule specifies (via the `u2` argument) that the next subproof of a `satlem` inference should prove clause `c3` under the assumption (`x`) that the simplified clause `c2` holds. Using an assumption here allows the proof to refer to the proven (simplified) clause without repeating its proof multiple times.

Efficient Proof-Checking. Our current C++ implementation of LFSC compiles a signature into an efficient C++ proof checker optimized for that signature. Compilation includes compiling side-condition functions like `simplify_clause` to efficient C++ code. The side-condition programming language is a simply typed first-order pure functional programming language, augmented with the limited imperative feature of setting marks on LFSC variables. For details of the optimizations implemented, see our previous work [6]. There, we demonstrated significant performance gains using the deferred resolution method, and significantly better proof-checking times than for two other proof checkers (CVC3+HOL and Fx7+Trew).

3 Compressing SMT Proofs Using RUP Inferences

Our goal now is to take advantage of recent advances in proof-checking for SAT to obtain further improvements in LFSC's runtime performance on SMT proofs. In the format described above, a proof consists of CNF conversion steps and lemmas, which contain theory reasoning steps and propositional reasoning steps. In most SMT implementations, propositional inferences are performed by the internal SAT solver in the form of conflict analysis or other procedures. Reverse Unit Propagation (RUP) has been proposed by van Gelder as an efficient propositional proof format [4]. The idea behind RUP is to check $F \vdash C$ by refuting $(F \cup \neg C)$ using only unit propagation. In this case, there is a proof of the empty clause using only *unit resolution*, which is like standard binary resolution except that one of the two resolved clauses is required to be a unit clause. Unit resolution is not refutation complete in general, but it has been shown to be complete when C is a conflict clause generated according to standard conflict-analysis algorithms [4]. In a proof based on RUP, only the clause C is recorded, and the sequence of such resolutions can be calculated from that clause. Thus, a long resolution proof of a RUP inference can be compressed to the concluded clause. It can happen, however, that writing down the clause itself takes more space in the proof than a short resolution proof would (a point worth exploring further in seeking smaller proofs).

3.1 Delegating Propositional Proofs

In principle, one could implement an RUP checker in the LFSC side condition language. This would require pure functional data structures for unit propagation, which would largely negate the benefits of the RUP proof format, which relies on the efficient unit propagation of modern SAT solvers. So instead, we delegate RUP proof checking to an external RUP checker; see Figure 3 for our work flow. The LFSC rules used to delegate the checking of propositional inferences from LFSC to the external RUP checker are presented in Figure 4. The external RUP checker confirms that certain *check clauses* follow by purely propositional reasoning from certain *assert clauses*. Assert clauses include the propositional clauses derived by CNF conversion from the original input formula; and also the boolean skeletons of all theory lemmas. In between some of these asserts, a proof can request that the proof checker confirm that a check clause follows by RUP

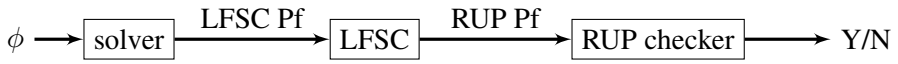

Figure 3: Workflow of New Proof System

```
(declare lemma (! c1 clause
               (! c2 clause
               (! z unit
               (! u1 (holds c1)
               (! r (^ (print_assert c1) z)
               (! u2 (! x (holds c1) (holds c2))
                 (holds c2))))))))
(declare check (! c2 clause
               (! z unit
               (! c1 clause
               (! r (^ (print_check c1) z)
               (! u (! x (holds c1) (holds c2))
                 (holds c2)))))))
```

Figure 4: LFSC Proof Rules for Delegating Checking of Propositional Inferences

from the (propositional) assert clauses, as well as previous check clauses.

The proof rule `lemma` (Figure 4) is used to assert a clause to the external RUP checker. It requires a proof (`u1`) that the asserted clause actually holds. The `check` rule then delegates checking that a clause `c1` follows from earlier clauses, including both asserted and already checked clauses, by purely propositional reasoning. Note that it does not require a proof of the clause `c1` which is being checked, since checking that this clause holds in the current logical context is being delegated to the external RUP checker. Both rules use side-condition functions (`print_assert` and `print_check`) to print out the clauses in question as either assert clauses or check clauses. Additionally, before printing any assert or check clauses, proofs in this signature must print an initial header, giving the number of propositional variables used.

Implementing Delegation. To support delegating propositional proofs, the LFSC compiler was modified to support printing of numbers, string literals, LFSC variables (used directly to encode propositional variables) from side-condition functions. To enable a very straightforward implementation, variables are printed out as their hexadecimal memory addresses. A simple post-processing phase, currently implemented by a short OCAML program, is used to map hexadecimal addresses to numbers starting with 1.

3.2 Propositional Proof Format

This section describes the proof format that the LFSC checker produces to delegate propositional reasoning to a RUP checker. It can be best explained by an example. Figure 5 shows an example propositional proof. We see the initial header, starting with `p`, specifying the maximum number of different variables that can appear in the file (here this is a loose bound). Then come assert clauses, which begin with `a` and are terminated with 0; and check clauses, which begin with `c` and are similarly terminated. The format of clauses is similar to the DIMACS format for CNF SAT problems. The example has four assert clauses and two check clauses intermixed. Obviously, the set of those assert clauses is refutable. It is not refutable, however, by unit resolution, because the assert clauses are all binary. Thus, the first check clause is necessary. The first

```
p 2
a 1 2 0
a 1 -2 0
c 1 0
a -1 2 0
a -1 -2 0
c 0
```

Figure 5: A Simple Propositional Proof

two assert clauses and the negation of the first check clause are refutable by resolving the negated check clause with the first two assert clauses and then resolving their resolvents. Now, the first check clause is verified. The last check clause, which is empty, simply asks if the entire clauses above it are refutable only using unit propagation without any more assumptions, which is true in this example.

3.3 The `clcheck` RUP Proof Checker

We implemented a RUP proof checker, called `clcheck` that supports the proof format explained above. Other proof checkers like `checker3` combined with `rupToRes`, which is used in the SAT competition, could be used with a proper translation. To the best of our knowledge, they do not support intermixed assertions and checks. Thus, assert clauses and check clauses have to be split into separate files. In SMT solvers, theory inferences and propositional inferences are naturally intermixed, and those theory inferences are asserted as clauses to be used in propositional inferences later on. We believe that intermixing assertions and checks in proofs allows concurrent processing of theory lemmas on the LFSC checker and propositional lemmas on the RUP checker, which can lead to more efficient proof checking on a modern multicore system. In our settings, the output of LFSC is directly streamed to `clcheck` using Unix pipes. So, while `clcheck` is checking a RUP inference, LFSC can check the next theory lemma at the same time.

A RUP inference $F \vdash C$ is verified as follows. First, for each literal in C, add a unit clause with the negation of that literal to the clause database. Now, the clause database has $F \cup \neg C$. Second, propagate all unit clauses in the database. If it leads to a conflicting clause, C is proved; otherwise, the inference is invalid. That can be also justified in terms of unit-resolution proof. Because every assignment is caused by a unit clause, which is the antecedent clause, the empty clause can be derived by applying unit resolution on each literal of C and that literal's antecedent clause. Finally, remove those unit clauses added in the first step and cancel all assignments. One can work more cleverly by avoiding redundancy. Instead of canceling all assignments, just cancel assignment only caused by $\neg C$ and, after C is verified, incrementally propagate the new unit clauses in $F \cup C$, which will be the new F for the next check. This approach is implemented in `clcheck`, which is written in C++ and which uses standard efficient data structures (in particular, watch lists for literals) for efficient unit propagation.

4 Preliminary Results

Our SMT solver `clsat` has been modified to generate proofs in the new format in addition to the original format. We have chosen 39 QF_IDL benchmarks that `clsat` solved in 900 seconds in the SMT competition 2009. Because `clsat` does not support the SMT-LIB 2.0 file format, they are in the SMT-LIB 1.2 format.

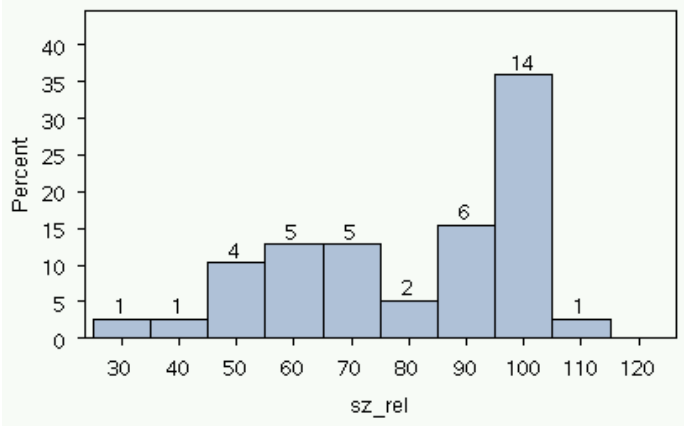

Figure 6: Distribution of Relative Proof Sizes

Table 1 (page 10) shows the results. The test machine had Intel Xeon X5650 2.67GHz CPU and 12GB of memory. Times (in seconds) are measured for solving and checking combined so that we can see how the new format improves the whole work flow, not just proof overhead. That measurement includes I/O overhead between the solver and checkers. The proof formats in comparison have different syntactic characteristics that may affect proof sizes. So, we wanted to compare the amount of information as the smallest number of bits needed to store the proof. That means you cannot modify the syntax to achieve a smaller proof. Instead of developing such a proof syntax, we used gzip-compression to approximate the amount of information in a proof. The table shows the gzip-compressed sizes (in bytes) of proofs. The uncompressed proof sizes did not change the conclusion. However, we believe the compressed sizes are more meaningful as data (especially, when we see the relative sizes). Note that one benchmark, `diamonds.18.5.i.a.u` did generate proofs, but failed to check in both formats due to memory overflow (uncompressed proof sizes reach 2GB in size).

Figure 6 shows the distribution of the relative sizes (ratios) of the new proofs. The horizontal axis is the relative size in percent (the size of new proof over the size of old proof times 100). And the vertical axis is the percentage of instances in each range. The number on each bar shows the number of instances in the range. For 14 benchmarks (accounting 35%), the new proof has almost the same size as the old counterpart (in the range of 95%-105%). However, there are a variety of compression ratios and mostly the new proofs are smaller or similar in size. One new proof is as small as 30% of the old counterpart. At the other extreme, there is one case that the new proof is 12% bigger than the old one. Figure 7 shows the correlation between the relative proof sizes and the relative proof checking times. For time comparison, we considered 11 benchmarks that take more than 1 second to solve and check on any system. Because small checking times have relatively big measurement errors, their relative times are not reliable. In the figure, each vertex represents a benchmark where its horizontal coordinate is the relative proof size and its vertical coordinate is the relative checking time. The figure shows a rough linear relationship between relative proof size and relative checking time along the regression line. The R^2 value of the regression is 0.7675. That can be summarized as the more a proof compresses in the new format, the more checking speeds up.

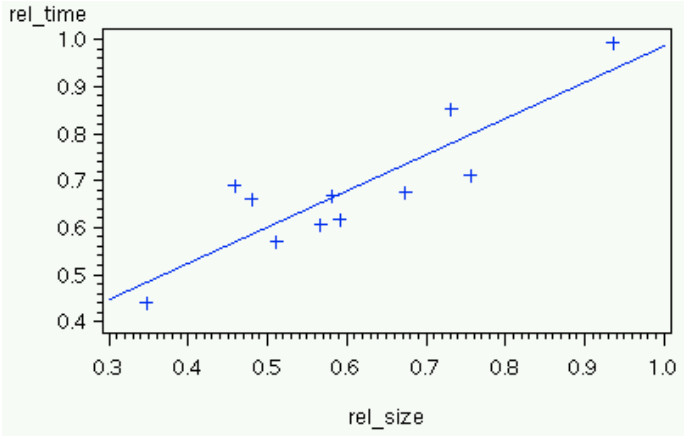

Figure 7: Correlation between Relative Checking Times and Relative Proof Sizes

5 Conclusion and Future Work

We have presented an approach for integrating an RUP checker for propositional proofs with the LFSC proof meta-language, based on delegation to an RUP checker. We have seen promising improvements over pure LFSC proof-checking, in both proof size and proof-checking time.

Improved LFSC implementation. As mentioned in the introduction, our team at The University of Iowa is implementing a new version of the LFSC checker, which we anticipate amplifying the benefits we have observed in our preliminary empirical results. Profiling the current version of LFSC on these benchmarks shows that at least in some cases, running the side-condition code (`simplify_clause` referenced in Figure 2) needed to check propositional resolution proofs is not taking a large part of the time for proof checking. Overhead in other parts of the proof checker outweighs this. Our new implementation is designed to take advantage of optimizations we described in earlier work on fast proof-checking for LF, the Edinburgh Logical Framework on which LFSC is based [8]. These optimizations are missing in the current LFSC checker. We anticipate they will lower the overhead of the rest of the proof-checking algorithm, and thus amplify the benefits of delegating propositional proofs to the RUP checker.

From `clcheck` to `vercheck`. In a separate line of research, the authors are implementing a statically verified modern SAT solver called `versat`. The specification we are establishing is that if the solver reports a set of input clauses unsatisfiable, then there exists a resolution proof of the empty clause from those input clauses. This resolution proof is not constructed at runtime. Rather, we prove that it is guaranteed to exist whenever the solver reports unsatisfiable. The `versat` solver uses standard efficient low-level data structures, based on mutable arrays, and implements standard modern SAT-solving techniques like conflict-driven clause learning, non-chronological backtracking, and watched literals.

Using the unit-propagation code in `versat`, we are implementing a trusted RUP checker called `vercheck`. The specification we are proving for this tool is that if it confirms an RUP proof of the kind described above, then the check clauses really do follow from the earlier check and assert clauses. Using `vercheck` will help mitigate the expansion of the trusted computing base incurred by delegating from LFSC. Our current LFSC C++ checker is around 6kloc C++. The new version currently in progress will be around 4.5kloc OCAML when complete. The `clcheck` solver is just under 1kloc C++. The trusted specification for `vercheck` is just 355 lines of GURU code (GURU is the research programming language we are using for implementation

and static verification of `versat` and `vercheck`). Also, the old signature for QF_IDL proofs from `clsat` is 870 lines of LFSC, while the new one is 795 lines. So using `vercheck`, the new approach based on delegation will only increase the number of lines of trusted code by 280 lines total, which seems a worthwhile price to pay for decreased proof size and improved proof-checking time.

References

[1] T. Bouton, D. Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE)*, pages 151–156, 2009.

[2] L. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In B. Konev, R. Schmidt, and S. Schulz, editors, *7th International Workshop on the Implementation of Logics (IWIL)*, 2008.

[3] P. Fontaine, S. Merz, and B. Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction (CADE)*, 2011. to appear.

[4] Allen Van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.

[5] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electr. Notes Theor. Comput. Sci.*, 144(2):43–51, 2006.

[6] D. Oe, A. Reynolds, and A. Stump. Fast and Flexible Proof Checking for SMT. In B. Dutertre and O. Strichman, editors, *Workshop on Satisfiability Modulo Theories (SMT)*, 2009.

[7] A. Stump and D. Oe. Towards an SMT Proof Format. In C. Barrett and L. de Moura, editors, *International Workshop on Satisfiability Modulo Theories*, 2008.

[8] M. Zeller, A. Stump, and M. Deters. Signature Compilation for the Edinburgh Logical Framework. In C. Schürmann, editor, *Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, 2007.

	solve+check time		compressed proof size	
benchmarks	old	new	old	new
BubbleSort_safe_blmc010	0.48	0.48	224122	220316
BubbleSort_safe_blmc016	0.94	0.94	406296	401250
CELAR7_SUB1	1.39	1.38	49754	46620
ckt_PROP0_tf_15	0.21	0.11	98277	59202
ckt_PROP1_tf_25	0.2	0.16	123252	99474
ckt_PROP2_tf_10	0.02	0.02	13762	13057
ckt_PROP5_tf_25	0.9	0.54	385940	252867
diamonds.18.5.i.a.u	Error	Error	112879127	126579755
DTP_k2_n35_c245_s19	1.32	0.89	363027	186938
DTP_k2_n35_c245_s5	2.51	1.66	709885	340784
DTP_k2_n35_c245_s6	3.36	2.32	924559	423192
FISCHER11-6-ninc	0.76	0.47	352386	241472
FISCHER13-1-ninc	0.03	0.03	25785	25642
FISCHER14-9-ninc	56.78	48.31	9851980	7214785
FISCHER6-2-ninc	0.03	0.03	24634	24146
FISCHER8-1-ninc	0.02	0.02	16079	15929
inf-bakery-invalid-2	0.01	0.01	7212	6690
int_incompleteness1	0	0	565	525
jobshop6-2-3-3-4-4-11	0.01	0.01	3923	3672
lpsat-goal-12	5.91	3.18	2124140	907239
lpsat-goal-15	19.48	11.09	5613533	2867306
lpsat-goal-2	0.05	0.05	43171	41749
lpsat-goal-8	0.92	0.6	486810	271645
plan-18.cvc	7.26	4.41	1570043	890322
plan-22.cvc	0.9	0.57	286853	190437
plan-30.cvc	5.71	2.51	1571073	545799
plan-33.cvc	23.6	14.55	4485851	2650662
plan-35.cvc	68.74	46.02	10937058	6369656
plan-9.cvc	0.06	0.06	37021	33638
PO2-2-PO2	0.01	0.01	10918	10691
PO2-6-PO2	0.04	0.05	40422	38543
PO4-10-PO4	2.64	1.78	1286666	867640
PO4-4-PO4	0.32	0.32	237111	232804
PO4-8-PO4	1.46	1.04	795994	602860
SelectionSort_safe_bgmc005	0.06	0.07	36991	36033
SelectionSort_safe_bgmc009	0.13	0.13	74919	72530
SortingNetwork4_safe_bgmc002	0	0	1833	1812
SortingNetwork8_safe_bgmc006	0.05	0.05	25317	25167
SortingNetwork8_safe_blmc006	0.18	0.17	75659	75482

* Measurement units: times in seconds, sizes in bytes

Table 1: Results of Old and New Proof Systems