Fast and Flexible Proof Checking for SMT

Duckki Oe, Andrew Reynolds, and Aaron Stump

Computer Science, The University of Iowa, USA

Abstract. Fast and flexible proof checking can be implemented for SMT using the Edinburgh Logical Framework with Side Conditions (LFSC). LFSC provides a declarative format for describing proof systems as signatures. We describe several optimizations for LFSC proof checking, and report experiments on QF_IDL benchmarks showing proof-checking overhead of 30% of the solving time required by our `clsat` solver.

1 Introduction

Correctness of the results of SMT solvers has long been a concern in the SMT community. While noteworthy efforts have been made to apply formal verification techniques to solver algorithms [4] and even actual solver code [6], solver implementors for now are mostly using runtime proof production to increase confidence when the solver reports unsatisfiability. CVC3, Fx7, and Z3 are example solvers which have the ability to emit refutation proofs [7, 8, 3]. We motivate the contribution of this paper by comparing the approaches to proof production taken by these three solvers.

Proof format. CVC3's proofs are in the format of the HOL Light theorem prover [5], while Fx7 and Z3 use custom proof formats. Fx7 uses a custom rewriting-based meta-language to enable concise and understandable expression of proof rules, to increase trust; while Z3 uses a particular natural deduction proof system with its own custom checker. For both Fx7 and Z3, introduction of new names during, for example, CNF conversion is an issue: Fx7 uses ad hoc restrictions on proof terms to ensure skolem constants are introduced properly, while Z3 uses quotation to introduce a variable named $\ulcorner\phi\urcorner$ for formula ϕ.

Proof checker size. The proof checkers used for CVC and Fx7 are quite small, with HOL Light based on a trusted core of 1500 lines of OCaml, and Fx7's C-language prover taking 1500 lines. (Line counts for a proof checker are not given for Z3 in [3].)

Proof checking overhead. We define proof-checking overhead to be the ratio of the time to produce and check the proof to the time required just for solving the benchmark without producing a proof. For Fx7, while proof checking time is reported as taking less than 1 second on average per benchmark for several thousand benchmarks of the AUFLIA division of SMT-COMP 2007 [1], proof-checking overhead is not reported. Since the AUFLIA benchmarks are generally solved very quickly by Fx7 (and other solvers), it is hard to draw conclusions about the overhead from the empirical data of [8]. No comparison of solving to

proof checking time is given for Z3 in [3], although slowdowns of 1.1x to 3x are reported due to proof generation (apparently not including proof checking) for five sample benchmarks from SMT-LIB [10]. For pigeon-hole SAT benchmarks in CVC3, numbers are reported showing that the time for HOL Light to certify a theorem by invoking CVC3 and checking the produced proof is (averaging the ratios for the numbers reported) around 40x slower than just running CVC3 and producing the proof [5].

Contributions. This paper demonstrates a flexible and efficient approach to confirming unsatisfiability results from SMT solvers by proof checking. We describe how to use the Edinburgh Logical Framework with Side Conditions (LFSC, Section 3) to encode SMT proofs for integer difference logic (QF_IDL). Advanced implementation techniques, including compilation (Section 6) and deferred resolution (Section 5), are used to check proofs efficiently which are produced by a modern SMT solver called `clsat`. **Our main result**: with our system, proof-checking overhead is on average 30% of solving time for the difficulty 0-3 unsatisfiable SMT-LIB QF_IDL benchmarks (Section 7). This improves significantly on the overhead reported in the literature for CVC3 and Z3 (we give a confirming empirical comparison in the appendix). Our empirical evaluation, in addition to being more thorough than those of the cited related work, is publicly available at `http://www.smtexec.org`. The trusted core is still reasonably small, indeed possibly verifiable: the proof checker is 3500 lines of C++. LF's native support for higher-order abstract syntax (HOAS) [9] enables a principled solution to the problem of introducing new names, which we demonstrate via a new approach to CNF conversion using *partial clauses* (Section 4).

2 Prover Architecture

In order to solve SMT formulas, we use an SMT solver/proof generator (CLSAT), and a proof verifier (LFSC). The solver CLSAT bested CVC3 in the QF_IDL division in SMT-COMP 2008, although solving many fewer benchmarks than the winning BARCELOGIC system [2]. CLSAT is built on top of a SAT solver that generates resolution proofs. In proof generating mode, all of the CNF conversion steps are recorded and recorded as lemmas. If the benchmark is unsatisfiable, CLSAT prints a sequence of deduction steps that derive the empty clause. CLSAT optimizes the size of proofs by pruning unnecessary CNF conversion steps and lemmas.

For proof checking, we use LFSC, a highly optimized type checker for the Edinburgh Logical Framework with Side Conditions. Various object logics can be succinctly declared via an LFSC *signature*. A signature defines the syntax for the logic's formulas, and its axioms and inference rules. This approach gives us the flexibility to define different proof formats based on different user-defined signatures. Additionally, LFSC extends LF by allowing the user to define computational side condition for inference rules, in a simple functional programming language. Our optimized approach to proof checking involves the compilation of these side conditions into the LFSC code base (Section 6).

3 LF with Side Conditions

Figure 1 gives typing rules for LFSC. Previous preliminary work gave only an informal semantics for LFSC [12]. The rules are based on the rules of so-called canonical forms LF [13]. There are judgment forms $\Gamma \Leftarrow t : T$ for checking that term t has type T in context Γ (where Γ, t, and T are inputs to the judgment); and $\Gamma \Rightarrow t : T$ for computing a type T for term t in context Γ (where Γ and t are inputs and T is output). The rules, read from conclusion to premises, determine deterministic type checking and type computation algorithms. We work up to a standard definitional equality, write x as a meta-variable for variables and constants declared in the context Γ, and use standard notation for capture-avoiding substitution (e.g., $[t/x]T$).

We focus here on the connection to side condition code, in the second rule for applications. Side conditions are of the form *run* C t, and are checked as part of checking an application, by testing whether or not evaluating C results in term t. C is a code term, with stateful operational semantics $\sigma; C \downarrow C'; \sigma'$, where C is the initial code term, C' the resulting value, σ is an initial state, and σ' the resulting state. States map LF variables x to a boolean mark, which is useful for operations like removing duplicate literals from a clause. For space reasons, the operational semantics and typing rules for code terms are relegated to the Appendix. The rules of Figure 1 enforce that bound variables cannot introduce side conditions, since the types of bound variables must be classified by the kind *type*, while types involving side conditions are classified by $type^c$.

We assume for all typing rules that contexts are well-formed in a standard sense, for which we omit rules. We have one additional requirement, not formalized in the figure. Suppose Γ declares a constant d whose type is of the form $\Pi x_1 : T_1. \cdots \Pi x_n : T_n.T$ of kind $type^c$, where T is of the form either c or $(c\ t_1\ \cdots\ t_m)$. Then neither c nor an application of c may be used as the domain of a Π-type. This is to ensure that applications requiring side condition checks never appear in types. Our implementation supports holes "_", which are arguments to applications whose values are determined by the types of later arguments. These are crucial to avoid bloating proofs with redundant information.

Define an operation $|T|$ on types by erasing side condition constraints from Π-abstractions in T. Also, define $|type^c|$ to be *type*. Extending this to contexts in the natural way, we may then easily prove the following by induction on the LFSC typing derivation:

Theorem 1. *If $\Gamma \vdash t : T$ in LFSC, then $|\Gamma| \vdash t : |T|$ in LF.*

4 Proof Encoding

Our LFSC signature is composed of several subsignatures for different aspects of the QF_IDL logic, totaling just 880 lines. To prove the unsatisfiablity of the input formula, ϕ, we derive false assuming ϕ. In LF, it can be stated as $\Gamma, f :$ (th_holds ϕ) $\vdash t :$ (holds *empty*) where Γ is the axiom of the object logic, f is the

$$\frac{}{\Gamma \Rightarrow \mathit{type} : \mathit{kind}} \qquad \frac{}{\Gamma \Rightarrow \mathit{type}^c : \mathit{kind}} \qquad \frac{\Gamma(x) = T}{\Gamma \Rightarrow x : T}$$

$$\frac{\Gamma \Rightarrow t_1 : \Pi x : T_1 . T_2 \quad \Gamma \Leftarrow t_2 : T_1}{\Gamma \Rightarrow (t_1\ t_2) : [t_2/x]T_2} \qquad \frac{\Gamma, x : T_1 \Leftarrow t : T_2}{\Gamma \Leftarrow \lambda x . t : \Pi x : T_1 . T_2}$$

$$\frac{\Gamma \Rightarrow t_1 : \Pi x : T_1 \mid \mathit{run}\ C\ t . T_2 \quad \Gamma \Leftarrow t_2 : T_1 \quad \emptyset \mid [t_2/x]C \downarrow [t_2/x]t \mid \sigma}{\Gamma \Rightarrow (t_1\ t_2) : [t_2/x]T_2}$$

$$\frac{\Gamma \Leftarrow T_1 : \mathit{type} \quad \Gamma, x : T_1 \Rightarrow T_2 : \kappa \quad \kappa \in \{\mathit{type}, \mathit{type}^c, \mathit{kind}\}}{\Gamma \Rightarrow \Pi x : T_1 . T_2 : \kappa}$$

$$\frac{\Gamma \Leftarrow T_1 : \mathit{type} \quad \Gamma, x : T_1 \Rightarrow T_2 : \mathit{type} \quad \Gamma, x : T_1 \Rightarrow C : T \quad \Gamma, x : T_1 \Rightarrow t : T}{\Gamma \Rightarrow \Pi x : T_1 | \mathit{run}\ C\ t . T_2 : \mathit{type}^c}$$

Fig. 1. Bidirectional Typing Rules for LFSC

assumption variable for the formula and t is the proof expression that derives false. About half the proof rules concern CNF conversion, which are designed to derive clauses out of given SMT formula. The rules give full control over the decision about which subformulas to rename by introducing a new variable, thanks to the notion of partial clause.

4.1 Partial Clauses

In order to convert an SMT formula like $\Phi_1 \vee \Phi_2 \vee \ldots \vee \Phi_n$ (Φ_i is an atomic formula) into a Boolean clause, we might wish to be able to declare n variables at once, for the subformulas. However, in LF, a rule can introduce only a fixed number of variables. Thus, it has to be done in multiple steps. To solve this problem and give provers choice of renaming, we introduce a new kind of formula called *partial clause*. Partial clauses store intermediate states between SMT formulas and Boolean clauses. Each partial clause is a tuple of a formula list and a clause. Its meaning (given by $[\![\cdot]\!]$) is the disjunction of all of elements in the formula list and the clause. Additionally, a list of partial clauses means their conjunction:

$$[\![(\phi_1, \cdots, \phi_n; l_1, \cdots, l_n)]\!] = \phi_1 \vee \ldots \vee \phi_n \vee l_1 \vee \ldots \vee l_n$$
$$[\![(\overline{\phi}_1; \overline{l}_1), \cdots, (\overline{\phi}_n; \overline{l}_n)]\!] = [\![(\overline{\phi}_1; \overline{l}_1)]\!] \wedge \ldots \wedge [\![(\overline{\phi}_n; \overline{l}_n)]\!]$$

4.2 CNF Conversion Rules

CNF conversion starts with one partial clause $(\phi, \cdot)$ in the partial clause database, where ϕ is the original formula. Then, other partial clauses can be added to the database using CNF conversion rules. Figure 2 gives several representative CNF conversion rules.

In the `rename_pos` and `decl_atom_pos` rules, new Boolean variables `v` are introduced. This is easily done in LF, due to its support for higher-order abstract syntax. The rule `decl_atom_pos` records the connection between `v` and ϕ,

$$
\begin{array}{lll}
\text{dist\_pos:} & (\phi_1 \wedge \phi_2, \overline{\phi}; C), \Pi \Rightarrow (\phi_1, \overline{\phi}; C), (\phi_1, \overline{\phi}; C), \Pi & \\
\text{dist\_neg:} & (\neg(\phi_1 \wedge \phi_2), \overline{\phi}; C), \Pi \Rightarrow (\neg\phi_1, \neg\phi_2, \overline{\phi}; C), \Pi & \\
\text{flat\_pos:} & (\phi_1 \vee \phi_2, \overline{\phi}; C), \Pi \Rightarrow (\phi_1, \phi_2, \overline{\phi}; C), \Pi & \\
\text{flat\_neg:} & (\neg(\phi_1 \vee \phi_2), \overline{\phi}; C), \Pi \Rightarrow (\neg\phi_1, \overline{\phi}; C), (\neg\phi_1, \overline{\phi}; C), \Pi & \\
\text{rename\_pos:} & (\phi, \overline{\phi}; C), \Pi \Rightarrow (\overline{\phi}; v, C), (\phi; \neg v), (\neg\phi; v), \Pi & (\text{v is new var}) \\
\text{decl\_atom\_pos:} & (\phi, \overline{\phi}; C), \Pi \Rightarrow (\overline{\phi}; v, C), \Pi & (v \mapsto \phi \text{ recorded})
\end{array}
$$

Fig. 2. Sample CNF conversion rules (Π is the rest of partial clauses)

while `rename_pos` adds new partial clauses for the definition of `v`. Conversion is continued until every partial clause is in the form of $(\cdot, \bar{l})$. Then, the `clausify` rule is used to convert a partial clause with empty formula part into a Boolean clause. Those rules can be seen as a non-deterministic top-down CNF conversion algorithm. Depending on the decision strategy, the actual algorithm could be the Tsetin algorithm, one that introduces a smaller number of variables, or one that produces a smaller number of clauses. However, CNF conversion proofs are just records of those decisions, and any top-down implementations can produce proofs regardless of their specific algorithms.

4.3 Theory Reasoning

When a theory solver detects a contradiction in the current model, it generates a clause that corrects the current model w.r.t its theory, which is in the Horn form: $\phi_1 \rightarrow \phi_2 \rightarrow \cdots \rightarrow \phi_n \rightarrow false$. This clause should be converted to a Boolean clause of the form: $\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_n$ where l_i is the (negated) Boolean variable that corresponds to (negated) ϕ_i. By witnessing $v_i \mapsto \phi_i$ or $v_i \mapsto \neg\phi_i$ recorded during CNF conversion, ϕ_i is replaced with $\neg v_i$ or v_i, respectively.

For QF_IDL logic, `idl_contra` rule is used to recognize a contradiction in the IDL theory: $x - x \leq c$, where x is a constant symbol and c is a negative integer value. And transitivity is the basic rule to derive a contradictory formula in IDL. Even in QF_IDL logic, SMT-LIB does not require atomic formulas to be strictly in the form of $x - y \leq c$. It could be in a simpler form, and one of the other equality or inequality predicates can be used instead. Thus, a number of normalization rules are provided to construct the strict IDL form. LFSC side conditions are used to verify the constant arithmetic constraints in each rule.

$$\frac{\Gamma \vdash x - x \leq c \quad \{c < 0\}}{\Gamma \vdash false}\ idl\_contra$$

$$\frac{\Gamma \vdash t_1 - t_2 \leq a \quad \Gamma \vdash t_2 - t_3 \leq b \quad \{a + b = c\}}{\Gamma \vdash t_1 - t_3 \leq c}\ idl\_trans$$

5 Deferred Resolution

Standard binary resolution with factoring computes a resolvent from clauses C_1 and C_2 by removing all positive occurrences of a given variable p from C_1, all

negative occurrences from C_2, and concatenating the resulting clauses. To keep clauses short, duplicate literals are then dropped. In our previous preliminary work, our LFSC proofs computed the resolvent in this way for each resolution inference [12]. Resolutions are used only during conflict clause generation, however, and are guaranteed by the conflict clause generation algorithm to be linear: one clause is always a member of the clause database. This paves the way for a more efficient functional computation of the final conflict clause.

We defer the computation of the resolvent until it is time to register the conflict clause as a lemma. When a resolution takes place, we employ a new clause of the form (literal, clause) that is interpreted to mean "clause with the given literal removed". Using this strategy, the resultant clause of a resolution can now be written simply as a union of two such pairs. The computation of such deferred resolutions is constant time per resolution.

The cost incurred with this approach is deferred to a final simplification step, in which the deferred resolution clause must be converted into a real clause. A formal description is shown in Figure 3. Clauses are either the empty clause (`cln`), a concatenation of a literal to a clause (`clc`), a removal of a literal from a clause (`clr`), or a concatenation of two clauses (`concat`). The state σ refers to a list of variables that have been marked for the sake of duplicate elimination. This algorithm returns a clause that does not contain `clr` or `concat` constructs. Since LFSC allows us to mark variables (but not other expressions), we use two passes: the first with b = false, and the second with b = true, computing $[\![[\![C]\!]^{\emptyset}_{ff}]\!]^{\emptyset}_{tt}$ for clause C. The first pass processes negative instances of literals in `clr` constructs, and the second positive instances. We structure resolutions so that the first clause given to `concat` is always from the clause database. Since the append operation (`++`) runs in time proportional to its first argument, this reduces the cost of appending the simplified clauses.

$$C ::= cln \mid clc\ L\ C \mid clr\ L\ C \mid concat\ C_1\ C_2$$

$$\begin{array}{ll}
[\![cln]\!]^{\sigma}_{b} & = cln \\
[\![(clc\ L\ C)]\!]^{\sigma}_{b} & = if(var(L) \in \sigma\ \&\&\ pol(L) = b)\ [\![C]\!]^{\sigma}_{b}\ else\ (clc\ L\ [\![C]\!]^{\sigma\ +\ var(L)}_{b}) \\
[\![(clr\ L\ C)]\!]^{\sigma}_{b} & = if(pol(L) = b)\ ([\![C]\!]^{\sigma\ +\ var(L)}_{b})\ else\ (clr\ L\ [\![C]\!]^{\sigma}_{b}) \\
[\![(concat\ C_1\ C_2)]\!]^{\sigma}_{b} & = if(b)\ ([\![C_1]\!]^{\sigma}_{b}\ +\!+\ [\![C_2]\!]^{\sigma}_{b})\ else\ (concat\ [\![C_1]\!]^{\sigma}_{b}\ [\![C_2]\!]^{\sigma}_{b}\ )
\end{array}$$

Fig. 3. Computing clauses from deferred resolutions

6 Optimizations for LFSC

Side condition compilation. Significant performance optimization has been achieved in LFSC by compiling side condition code to C++, and including it with the rest of the LFSC code base. This improves greatly upon simply interpreting

side condition code, as done in our previous work [12]. LFSC reads in the side condition code and emits new C++ code for checking it, based on standard compilation techniques for functional programming. LFSC is then recompiled with the newly emitted code. A diagram of this process is shown in Figure 4.

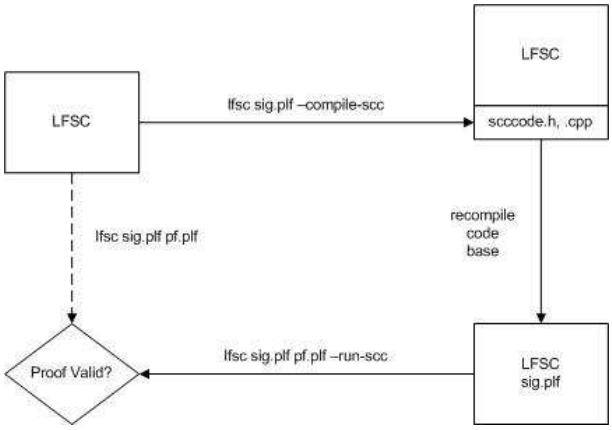

Fig. 4. Proof checking architecture using LFSC

Spine form applications. We have found that using spine form for the application construct, where arguments to function and program calls reside on the first level of the syntax tree, as opposed to left-nested applicative form leads to significant performance improvement in LFSC, in particular during the execution of compiled side condition code.

Path compression for holes. When a hole "_" is filled in during type checking, a pointer is set from the hole to the term t filling it. Something similar happens during β-reduction, when a pointer is set from a variable to a term t being substituted for it. When these pointers are set, as an optimization we make sure to follow all such pointers from t, if t is itself a hole or variable. This also contributes to a significant performance improvement.

Incremental checking. As described in previous work [11], LFSC interleaves parsing and checking of proofs. This results in significant performance improvements over parsing the whole proof into memory and then checking it, and opens the way for checking proofs too large to fit into main memory.

Simple type caching. Whenever a Π-abstraction is created within LFSC, the program will calculate whether or not its input variable is free within its body. This result is stored as a bit in the data field of the PI abstraction expression, and is used to determine if a simply typed function is being applied while type checking. This enables several optimizations, including the possibility of not constructing the argument term in memory (see [11]).

7 Empirical Results

Solver	Score	Unknown	Timeout	Time 1	Time 2
clsat r453 (w/o proof)	542 / 622	50	30	20168.7s	31843.6s
clsat+lfsc r591 (optimized)	538 / 622	51	33	23741.4s	41420.8s
clsat+lfsc r453 (unoptimized)	485 / 622	58	79	52373.8s	n/a

Table 1. Summary of results (timeout: 1800 seconds)

Unsatisfiable QF_IDL benchmarks of difficulty 0 through 3 were used for this test. The experiments were performed on the SMT-EXEC service (`http://www.smtexec.org`), and the results are publicly available in two jobs: `clsat-lfsc-2009` and `r591` (cf. a combined job `clsat-lfsc-2009.3`, still running at the time of writing). A timeout of 1800 was used. The results for CLSATr453 are for solving only, with no proof production or proof checking. We consider both unoptimized and fully optimized LFSC configurations. The unoptimized configuration does not use any of the optimizations of Section 6 except incremental checking. The optimized configuration uses all optimizations and deferred resolution. The "Score" column gives the number of benchmarks each configuration finished successfully. The "Unknown" column gives the number of benchmarks each configuration failed before timeout. The "Time 1" column gives times taken for 485 benchmarks that all of the configurations could solve. The "Time 2" column gives times taken for 538 benchmarks that the fast two configurations could solve. The optimized LFSC showed a 2.2x improvement over the unoptimized version for the 485 benchmarks. And the overhead of proof generation and proof checking over solving was only 17.71% on average for those benchmarks. For the 538 benchmarks, the overhead was 30.08% on average. (See Appendix E for comparison with other systems.)

Figure 5, generated by SMT-EXEC, compares CLSAT+LFSC to CLSAT (without proof recording and printing). It turned out that the proofs from certain families of benchmarks are more difficult to verify than those of other families. Those families are fischer, diamonds, planning and post_office. The worst case is `diamonds.11.3.i.a.u.smt` as its proof generation + checking time was 2.2 seconds while its solving time was 0.2 seconds. However, the figure also shows that the more difficult the benchmarks are, the closer the time for CLSAT+LFSC is to the time for just CLSAT. Figure 6 shows the very large improvement given by the proof checking optimizations of Sections 5 and 6 over unoptimized proof checking. Note that these optimizations had different levels of effectiveness on improving the proof checking time (see Appendix D).

Acknowledgments. This work was partially supported by NSF award CCF-0841554. Many thanks to Morgan Deters for answering bug reports and feature requests for SMT-EXEC during the course of this work.

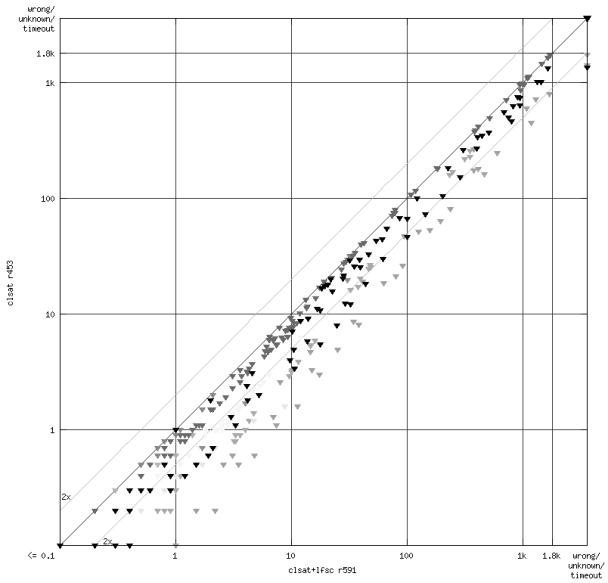

Fig. 5. Solving versus solving + optimized proof checking

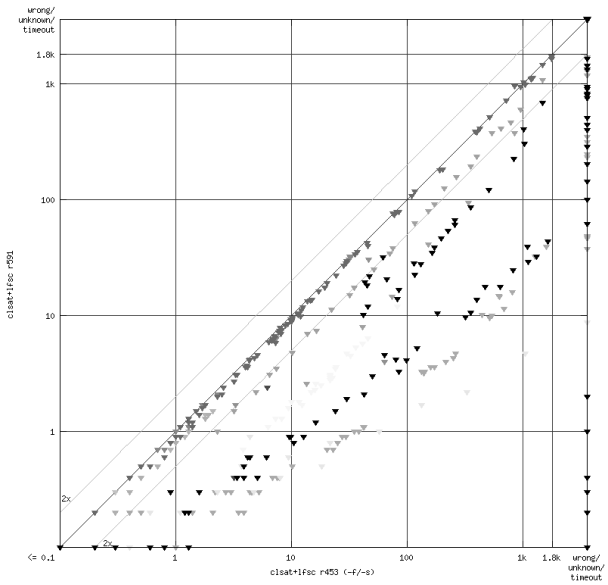

Fig. 6. Solving + optimized proof checking versus solving + unoptimized proof checking

References

1. C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and Results of the 3rd Annual Satisfiability Modulo Theories competition (SMT-COMP 2007). *International Journal of Artificial Intelligence Tools*, 2008. to appear.
2. M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT Solver. In A. Gupta and S. Malik, editors, *20th International Conference on Computer Aided Verification (CAV)*, pages 294–298, 2008.
3. 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.
4. J. Ford and N. Shankar. Formal verification of a combination decision procedure. In A. Voronkov, editor, *18th International Conference on Automated Deduction*, 2002.
5. John Harrison. HOL Light: A Tutorial Introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 265–269, 1996.
6. S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.
7. 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.
8. M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
9. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.
10. S. Ranise and C. Tinelli. The SMT-LIB Standard, Version 1.2, 2006. Available from the "Documents" section of http://combination.cs.uiowa.edu/smtlib.
11. A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.
12. 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.
13. K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2002.

A LFSC Typing Rules for Code Terms

Figure 7 gives typing rules for LFSC code terms. These are completely standard for a pure monomorphic functional programming language. We use only our type computation judgment here. We write N for any arbitrary precision integer, and use several arithmetic operations on these; others can be easily modularly added. Function applications are required to be simply typed. In the typing rule for pattern matching expressions, patterns P must be of the form c or $(c\ x_1\ \cdots x_m)$,

where c is a constructor, not a bound variable (we do not formalize the machinery to track this difference). In the latter case, $\mathit{ctxt}(P) = \{x_1 : T_1, \ldots x_n : T_n\}$, where c has type $\Pi x_1 : T_1. \cdots x_n : T_n.T$. We sometimes write $(\mathit{do}\ C_1\ C_2)$ as an abbreviation for $(\mathit{let}\ x\ C_1\ C_2)$, where $x \notin \mathit{FV}(C_2)$.

$$\frac{\Gamma(x) = T}{\Gamma \Rightarrow x : T} \qquad \frac{}{\Gamma \Rightarrow N : \mathit{mpz}}$$

$$\frac{\Gamma \Rightarrow t_1 : \mathit{mpz} \quad \Gamma \Rightarrow t_2 : \mathit{mpz}}{\Gamma \Rightarrow t_1 + t_2 : \mathit{mpz}} \qquad \frac{\Gamma \Rightarrow t : \mathit{mpz}}{\Gamma \Rightarrow -t : \mathit{mpz}}$$

$$\frac{\Gamma \Rightarrow C_1 : T' \quad \Gamma, x : T' \Rightarrow C_2 : T}{\Gamma \Rightarrow (\mathit{let}\ x\ C_1\ C_2) : T} \qquad \frac{\Gamma \Rightarrow C_1 : \mathit{mpz} \quad \Gamma \Rightarrow C_2 : T \quad \Gamma \Rightarrow C_3 : T}{\Gamma \Rightarrow (\mathit{ifneg}\ C_1\ C_2\ C_3) : T}$$

$$\frac{\Gamma \Rightarrow C : T}{\Gamma \Rightarrow (\mathit{markvar}\ C) : T} \qquad \frac{\Gamma \Rightarrow t_1 : \Pi x : T_1. T_2 \quad \Gamma \Rightarrow t_2 : T_1 \quad x \notin \mathit{FV}(T_2)}{\Gamma \Rightarrow (t_1\ t_2) : T_2}$$

$$\frac{\Gamma \Rightarrow T : \mathit{type}}{\Gamma \Rightarrow (\mathit{fail}\ T) : T} \qquad \frac{\Gamma \Rightarrow C_1 : T' \quad \Gamma \Rightarrow C_2 : T \quad \Gamma \Rightarrow C_3 : T}{\Gamma \Rightarrow (\mathit{ifmarked}\ C_1\ C_2\ C_3) : T}$$

$$\frac{\Gamma \Rightarrow C : T \quad \forall i \in \{1, \ldots, n\}.(\Gamma \Rightarrow P_i : T \quad \Gamma, \mathit{ctxt}(P_i) \Rightarrow C_i : T')}{\Gamma \Rightarrow (\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_n\ C_n)) : T'}$$

Fig. 7. Typing Rules for Code Terms

B LFSC Operational Semantics for Code Terms

The big-step operational semantics for LFSC code terms is given in Figure 8. States σ map LF variables x to a boolean mark. We write N for any arbitrary precision integer, and use several arithmetic operations on these; others can be easily modularly added. We use standard notation for functional updating $(\sigma[x \mapsto v])$. The last rule of the figure is for applications of top-level recursively defined functions p, whose singly recursive definition is given by $\mathit{def}(p)$ (we omit the straightforward typing rules for these definitions). If no rule applies, evaluation and hence type checking fails. This can happen for example, when evaluating an explicit $(\mathit{fail}\ T)$ code term (typing rule in the Appendix) or if a pattern match fails. We do not enforce termination of side condition programs, nor do we attempt to provide facilities for formal reasoning about the behavior of such programs.

$$\frac{}{\sigma_1; N \downarrow N; \sigma_1} \qquad \frac{}{\sigma_1; x \downarrow x; \sigma_1} \qquad \frac{\sigma_1; C \downarrow x; \sigma_2}{\sigma_1; (\mathit{markvar}\ C) \downarrow x; \sigma_2[x \mapsto \neg\sigma_2(x)]}$$

$$\frac{\sigma_1; t \downarrow N; \sigma_2 \quad N' = -N}{\sigma_1; -t \downarrow N'; \sigma_2} \qquad \frac{\sigma_1; t_1 \downarrow N_1; \sigma_2 \quad \sigma_2; t_2 \downarrow N_2; \sigma_3 \quad N_1 + N_2 = N}{\sigma_1; t_1 + t_2 \downarrow N; \sigma_3}$$

$$\frac{\sigma_1; C_1 \downarrow N; \sigma_2 \quad N < 0 \quad \sigma_2; C_2 \downarrow C_2'; \sigma_3}{\sigma_1; (\mathit{ifneg}\ C_1\ C_2\ C_3) \downarrow C_2'; \sigma_3} \qquad \frac{\sigma_1; C_1 \downarrow N; \sigma_2 \quad N \geq 0 \quad \sigma_2; C_3 \downarrow C_3'; \sigma_3}{\sigma_1; (\mathit{ifneg}\ C_1\ C_2\ C_3) \downarrow C_3'; \sigma_3}$$

$$\frac{\sigma_1; C_1 \downarrow x; \sigma_2 \quad \sigma_2(x) \quad \sigma_2; C_2 \downarrow C_2'; \sigma_3}{\sigma_1; (\mathit{ifmarked}\ C_1\ C_2\ C_3) \downarrow C_2'; \sigma_3} \qquad \frac{\sigma_1; C_1 \downarrow x; \sigma_2 \quad \neg\sigma_2(x) \quad \sigma_2; C_3 \downarrow C_3'; \sigma_3}{\sigma_1; (\mathit{ifmarked}\ C_1\ C_2\ C_3) \downarrow C_3'; \sigma_3}$$

$$\frac{\sigma_1; C_1 \downarrow C_1'; \sigma_2 \quad \sigma_2; [C_1'/x]C_2 \downarrow C_2'; \sigma_3}{\sigma_1; (\mathit{let}\ x\ C_1\ C_2) \downarrow C_2'; \sigma_3} \qquad \frac{\forall i \in \{1, \ldots, n\}, (\sigma_i; C_i \downarrow C_i'; \sigma_{i+1})}{\sigma_1; (x\ C_1\ \cdots\ C_n) \downarrow (x\ C_1'\ \cdots\ C_n'); \sigma_{n+1}}$$

$$\frac{\sigma_1; C \downarrow (c\ C_1'\ \cdots\ C_k'); \sigma_2 \quad P_i = (c\ x_1\ \cdots\ x_k) \quad \sigma_2; [C_1'/x_1, \ldots, C_k'/x_k]C_i \downarrow C'; \sigma_3}{\sigma_1; (\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_n\ C_n)) \downarrow C'; \sigma_3}$$

$$\frac{\begin{array}{l}\forall i \in \{1, \ldots, n\}, (\sigma_i; C_i \downarrow C_i'; \sigma_{i+1}) \\ \mathit{def}(p) = \lambda x_1. \cdots \lambda x_n. C \qquad \sigma_{n+1}; [C_1'/x_1, \ldots, C_n'/x_n]C \downarrow C'; \sigma_{n+2}\end{array}}{\sigma_1; (p\ C_1\ \cdots\ C_n) \downarrow C'; \sigma_{n+2}}$$

Fig. 8. Operational Semantics of Code

C Compiled Side Condition Code

An example of a side condition and the corresponding compilable code is displayed below. Note that the C++ code here has been simplified for the sake of clarity.

```
(program append ((c1 clause) (c2 clause)) clause
  (match c1 (cln c2) ((clc l c1') (clc l (append c1' c2)))))
```

```
Expr* f_append( Expr* c1, Expr* c2 ){
   Expr* e0;
   Expr* e1 = c1->get_head();
   static Expr* e2 = symbols->get("cln");
   static Expr* e3 = symbols->get("clc");
   if( e1==e2 ){
      e0 = c2;
   }else if( e1==e3 ){
      Expr* l = c1->kids[1];
      Expr* c1h = c1->kids[2];
      Expr* e4;
      e4 = f_append( c1h, c2 );
      static Expr* e5 = symbols->get("clc");
```

```
      e0 = new CExpr( APP, e5, l, e4 );
   }else{
      std::cout << "Could not find match for expression";
   }
   return e0;
}
```

This outputted code sequence is a reflection of what code would have been executed by the interpreter if LFSC had not utilized compiled side condition code. However, the code is now unrolled in a form that does not require LFSC to traverse the syntax tree of the side condition, and rather can run the compiled code directly.

D Results of Individual Optimizations

CLSAT+LFSC configurations using individual optimizations (deferred resolution, compiled scc, path compression) have been compared over the same set of benchmarks described in Section 7.

Solver	Score	Unknown	Timeout	Total Time
clsat+lfsc (unoptimized)	485 / 621	57	79	52373.8s
clsat+lfsc (fully optimized)	538 / 621	50	33	41420.8s
clsat+lfsc (deferred resolution)	526 / 621	58	37	49603.7s
clsat+lfsc (compiled scc)	515 / 621	58	48	44362.2s
clsat+lfsc (path compression)	511 / 621	57	53	52035.3s

At the time of writing, one of the benchmarks was not finished on the SMT-EXEC job. So, shown are the results of 621 benchmarks finished. It shows that the fully optimized version is better than any of the individual optimizations, especially in the number of unknown results. In terms of scores, deferred resolution was the winner among the individual optimizations. However, the other optimizations also showed quite a bit of improvements over the unoptimized version.

E Comparison with other SMT proof checkers

We consider CVC3 and Fx7 that are known to be able to produce proofs and verify them. For CVC3, cvchol was used. cvchol is an extension to HOL light that embeds CVC3's functionality and verifies proofs returned by the embeded CVC3. For Fx7, the C version of trew was used to verify proofs that generated by Fx7.

cvchol required OCaml runtime that was not available on the SMT-EXEC machines at the time of writing. So, it was not possible to compare these solvers on SMT-EXEC. Instead, we sampled 25 unsatisfiable benchmarks from SMT-COMP 2008 that CVC3 solved in 300 seconds. Each solver and combination with

proof checker were tested against those benchmarks. The tests were performed on a Athlon 64 X2 3800+ machine with 2GB of memory running Ubuntu linux 9.04. A timeout of 600 was used. Note that the results here may be different from that of SMT-COMP 2008 because unscrambled benchmarks were used.

Benchmarks	clsat	clsat+lfsc	cvc3	cvchol	fx7	fx7+trew
01.100.graph.smt	0.29	0.38	1.48	1.25	1.93	FAIL
03.700.graph.smt	22.37	24.46	37.01	T/O	ERR	N/A
03.800.graph.smt	7.44	9.88	50.9	T/O	ERR	N/A
07.700.graph.smt	478.36	484.11	38.88	T/O	ERR	N/A
07.800.graph.smt	8.11	11.02	55.02	T/O	ERR	N/A
10.700.graph.smt	11.22	13.59	43.32	T/O	ERR	N/A
11.500.graph.smt	5.77	6.94	20.31	T/O	237.65	ERR
17.400.graph.smt	102.24	102.85	14.39	460.51	T/O	N/A
17.800.graph.smt	8.68	11.09	59.85	T/O	ERR	N/A
18.600.graph.smt	5.82	7.44	33.89	T/O	81.66	ERR
20.300.graph.smt	1.09	1.45	9.51	131.21	34.1	ERR
20.700.graph.smt	8.21	10.15	44.79	T/O	ERR	N/A
22.600.graph.smt	3.9	5.25	32.88	T/O	25.36	ERR
24.400.graph.smt	9.4	9.98	12.53	318.79	T/O	N/A
24.500.graph.smt	382.71	384.22	20.83	T/O	T/O	N/A
27.700.graph.smt	6.68	8.69	39.94	T/O	ERR	N/A
27.800.graph.smt	8.98	11.73	56.34	T/O	ERR	N/A
28.600.graph.smt	5.33	6.61	30.28	T/O	88.84	ERR
31.200.graph.smt	0.73	0.96	3.43	17.81	12.79	ERR
FISCHER10-10-ninc.smt	221.62	294.94	2.39	SEGV	ERR	N/A
FISCHER11-10-ninc.smt	293.96	383.3	2.77	SEGV	ERR	N/A
FISCHER11-11-ninc.smt	T/O	T/O	3.02	SEGV	ERR	N/A
FISCHER12-10-ninc.smt	282.41	376.48	2.87	SEGV	ERR	N/A
FISCHER12-11-ninc.smt	T/O	T/O	3.17	SEGV	ERR	N/A
FISCHER13-10-ninc.smt	310.04	403.59	3.14	SEGV	ERR	N/A

Fx7 competed in AUFLIA on SMT-COMP 2007. Even though QF_IDL is a subset of AUFLIA, Fx7 was not meant to compete in QF_IDL. In our experiments, Fx7 failed to either solve or produce proofs (reported as ERR on the table) for most cases. There was one case (`01.100.graph.smt`) that Fx7 succeded to produce a proof, but its proof checker didn't accept the proof. On the other hand, cvchol could verify more benchmarks, but the fischer benchmarks caused segmentation faults. Note that because cvchol runs on top of HOL light, we subtracted the overhead of loading HOL light from the entire running time of cvchol, so as to measure only the time for solving and proof checking.

F Small LFSC Examples

This section lists several small example proofs in our LFSC signature. We use ascriptions of the form `(: A t)` to direct LFSC to check that term `t` has type `A`.

Finally, we use a type-computing lambda abstraction $\lambda x : A.t$, written concretely in LFSC as `(% x A t)`.

```
(% p (term Bool)
(% @f (th_holds (and (p_app p) (not (p_app p))))
(: (holds cln)
(start _ @f
(\ @f0
(dist_pos _ _ _ _ @f0
(\ @f1 (\ @f2
(decl_atom_pos _ _ _ @f1
(\ @v0 (\ @a0 (\ @f3
(clausify _ @f3
(\ @x0
(subst_atom_neg _ _ _ _ @f2 @a0
(\ @f4
(clausify _ @f4
(\ @x1
(satlem _ _ _ (R _ _ @x0 @x1 @v0)
(\ @done @done))
))))))))))))))))))
```

Fig. 9. An example proof in LFSC of $p \wedge \neg p \Rightarrow$ false

```
(% x (term Int)
(% y (term Int)
(% z (term Int)
(% @f (th_holds (and (<= (- x y) (an_int (~ 1)))
                (and (<= (- y z) (an_int (~ 2)))
                     (<= (- z x) (an_int (~ 3))))))
(: (holds cln)
(start _ @f
(\ @f0
(dist_pos _ _ _ _ @f0
(\ @f1 (\ @f2
(decl_atom_pos _ _ _ @f1
(\ @v0 (\ @a0 (\ @f3
(clausify _ @f3
(\ @x0
(dist_pos _ _ _ _ @f2
(\ @f4 (\ @f5
(decl_atom_pos _ _ _ @f4
(\ @v1 (\ @a1 (\ @f6
(clausify _ @f6
(\ @x1
(decl_atom_pos _ _ _ @f5
(\ @v2 (\ @a2 (\ @f7
(clausify _ @f7
(\ @x2
(satlem _ _ _
  (R _ _ @x0
    (R _ _ @x1
      (R _ _ @x2
        (assume_true _ _ _ @a0 (\ @h0
        (assume_true _ _ _ @a1 (\ @h1
        (assume_true _ _ _ @a2 (\ @h2
          (idl_contra _ _
            (idl_trans _ _ _ _ _ _ @h0
            (idl_trans _ _ _ _ _ _ @h1
                                  @h2)))))))))
        @v2) @v1) @v0)
(\ @done @done))
))))))))))))))))))))))))))))))))
```

Fig. 10. A small QF_IDL proof