Proof Checking Technology for Satisfiability Modulo Theories

Aaron Stump

Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
astump@acm.org

Abstract

A common proof format for solvers for Satisfiability Modulo Theories (SMT) is proposed, based on the Edinburgh Logical Framework (LF). Two problems arise: checking very large proofs, and keeping proofs compact in the presence of complex side conditions on rules. *Incremental checking* combines parsing and proof checking in a single step, to avoid building in-memory representations of proof subterms. LF with Side Conditions (LFSC) extends LF to allow side conditions to be expressed using a simple first-order functional programming language. Experimental data with an implementation show very good proof checking times and memory usage on benchmarks including the important example of resolution inferences.

Key words: Edinburgh LF, incremental checking, Satisfiability Modulo Theories, LF with Side Conditions

1 Introduction

At the heart of several current automated verification methods are automated reasoning tools for solving the problem of Satisfiability Modulo Theories (SMT). SMT solvers combine implementations of special-purpose reasoning algorithms for theories like arithmetic or arrays, with propositional (SAT) or first-order reasoners. SMT tools are increasingly used in *algorithmic verification*, where verification problems are translated into SMT formulas, often very large, to be solved (e.g., [2]). Verification based on interactive theorem proving relies on similar automated reasoning algorithms. Usability of SMT solvers

[1] This work supported by the U.S. National Science Foundation under grant number CNS-0551697.

for applications has seen a big improvement with the widespread adoption of a standard input language [8]. This language has been devised as part of the SMT-LIB initiative, currently coordinated by Clark Barrett, Silvio Ranise, the author, and Cesare Tinelli.

For algorithmic verification, trustworthiness of the results of the solver is hard to establish without producing and checking proofs. This is due to the complexity of modern SMT solvers, with codebases often between 50k to 100k lines of C++. These tools exhibit bugs, and are not trustworthy enough for verification of critical systems. One approach to addressing this problem is for SMT solvers to emit independently checkable evidence of the results they report. For formulas reported unsatisfiable, this evidence takes the form of a refutation proof of the formula. Since the relevant proof checking algorithms and implementations are much simpler than those for SMT solvers, checking a proof provides a more trustworthy confirmation of the solver's result. Interactive theorem proving can also benefit from proof producing SMT solvers, as shown recently for Isabelle/HOL with the haRVey solver [4]. Complex subgoals can be discharged by the SMT solver, and the proof it returns can be checked or reconstructed (subject to resource limitations) to confirm the result without trusting the solver.

1.1 Using LF for SMT

Proof production is currently not widely supported among SMT solvers. One of the main reasons for this is the lack of a common proof format. Such a format, supported by a proof checker and defined as part of the SMT-LIB standard, will facilitate implementation of proof production in SMT solvers by providing a common target for proof production. A common proof format is also a critical first step to interfacing SMT solvers with theorem provers. Such a format will serve as an intermediate language, which can then be translated into the formats of particular theorem provers.

The Edinburgh Logical Framework (LF) has been used in several prominent projects in the domains of proof-carrying code and proof-carrying authentication [3,1]. A central concern in these applications has been minimizing proof size, since the use model requires proofs to be transmitted over a network to convince skeptical parties that certain operations are safe or permitted [11,5]. In the works just cited, this is done essentially by a protocol where the skeptical party is to be convinced not by a proof encoded in LF, but by a logic program, where successful runs of the program can be checked (by the skeptical party) to correspond to LF-encoded proofs of the desired safety property.

The same qualities that make LF attractive for proof-carrying code applications make it so as a proof format for SMT solvers. These qualities are centrally flexibility and support for higher-order abstract syntax (HOAS) [6].

The advantages of HOAS are well-known for encoding syntax with binding, such as is found in SMT formulas, which can make use of first-order quantifiers. Flexibility is very important for SMT, since the variety of logical theories supported by SMT solvers, the variety of deductive systems used to describe the solving algorithms, and the relatively early stage of development of the field mean that it would be very difficult if not practically impossible to design a single set of inference rules that would be a good target for all solvers. A first step to a common logic is to have a common meta-logic in which solver implementors can describe their axioms and rules.

But using LF for SMT solvers poses engineering challenges not encountered in the application to proof-carrying code. Particularly, the problem is not so much to minimize proof size for transmission over a network, but to cope with the very large proofs (easily hundreds of megabytes, if not gigabytes) that can be produced, even for relatively short runs of a modern SMT solver. In this paper, we address two problems related to checking very large proofs: how to bound memory usage (during proof checking) and how to avoid bloating inferences with proofs of side conditions. This paper's solutions to these problems are *incremental checking* and *LF with Side Conditions* (LFSC).

1.2 Techniques for Checking Large Proofs

A very standard approach to LF checking, and to many other language processing problems, is to parse textual input into an abstract syntax tree (AST), and then process the AST. This approach is a non-starter for checking very large proofs, for several reasons. First, we should not count on being able to fit the AST into main memory. Using the experimental setup described below, we can easily generate 100M proofs using a prototype proof-producing SAT solver in a relatively short time, on the order of tens of minutes. Longer runs, or runs using an SMT solver, which perform theory-specific inferences in addition to the propositional ones recorded by the SAT solver, may easily add a factor of 10 or 100 to the proof size. Storing proofs of this size in main memory may then prove quite difficult. A second problem with checking very large proofs arises when the proofs are deeply nested. In this case, a naive recursive implementation of the proof checking algorithm, or even the parser, can lead to stack overflow.

The solution to the first part of this problem is to interleave parsing and proof checking, thus enabling incremental proof checking. The entire proof need never be read into main memory. Furthermore, the proof can be checked by the proof checker while it is being produced by the SMT solver. The SMT solver emits the proof to its standard output channel, which is piped into the proof checker's standard input channel. This architecture makes effective use of two cores (one for the solver, another for the proof checker) of modern multicore computers. In addition, it opens up the possibility that the proof itself need never be written to disk. The proof checker is acting as an online double checker of the results of the SMT solver. Incremental checking is complicated by the use of dependent types in LF, and by bidirectional type checking. We see below how these are accommodated. To solve the stack overflow problem, we will see below how to structure proofs and slightly modify the type checking algorithm so that tail recursion can be used in critical places.

The problems of dealing with very large proofs are exacerbated if inferences must contain proofs of usually tacit side conditions. Consider the resolution proof rule. This rule, which historically has a central place in automated reasoning, is used critically in state-of-the-art SAT and SMT solvers, in particular during clause learning [13]. We consider here a relatively simple form of (propositional) resolution, namely binary resolution with factoring. Supporting more complex rules such as hyperresolution should be possible by extending the approach here, but that must remain to future work. Binary resolution with factoring says that from clauses C_1 and C_2, where C_1 contains variable v positively and C_2 contains it negatively; we may conclude the resolvent of C_1 and C_2, computed as follows. Remove all the positive occurrences of v from C_1, all the negative ones from C_2, and append the resulting clauses to get the resolvent. Optionally (without affecting completeness), we may subsequently remove other duplicate literals in the resolvent.

To encode binary propositional resolution with factoring in pure LF, we must insist that each resolution inference comes with a proof of a side condition showing that C_1 and C_2 resolve as just described to give the resolvent, which the inference proves. That proof may be a trace of the computation of the resolvent, or perhaps evidence based on a more declarative view of the relationship between the resolvent and C_1 and C_2. But there is no obvious way to reduce its size from $O(|C_1| + |C_2|)$ to a constant. And hence, the size of resolution proofs will be completely dominated by the size of the proofs of their side conditions.

To deal with this problem, we adapt the solution used by previous authors to decrease proof size [11,5]. There, checking a proof was replaced with running a verified logic program. Here, we seek a solution enabling a spectrum of methods from completely declarative (pure LF) to completely computational (as in the works just cited) proof checking. The proposed approach is called LF with Side Conditions (LFSC). Encoded inference rules may stipulate side conditions using code written in a very simple functional programming language. Whenever the encoded inference rule is applied (to all its arguments, as we will insist), the side condition is checked by running the given side condition code. If running the code succeeds, then the side condition check does, too. If the code fails (as it may explicitly do, or do by failure of pattern matching), then the side condition check fails and the LFSC type checker rejects the application of the rule.

In the rest of this paper, we present the ideas of incremental checking (Section 2) and LF with Side Conditions (Section 3) in more detail, and present promising empirical results, obtained with a prototype incremental LFSC checker.

2 Incremental Checking

The central idea of incremental checking is to interleave parsing and proof checking. We keep track of when we need to build an abstract syntax tree (AST) for textual input. The crucial observation is that if we have determined that we do not need to create the AST for an application $(t_1\ t_2)$, where $t_1 : \Pi x : A. B$, then we need to create an AST for t_2 only if $x \in FV(B)$. If $x \notin FV(B)$, then we just check the textual input for the argument t_2 without constructing it.

The incremental checking algorithm thus must distinguish between a mode of operation in which ASTs must be constructed for textual input ("creating" mode) and a mode in which the AST need not be constructed ("non-creating" mode). In addition, we make use of bidirectional checking, as advocated for so-called canonical forms LF [10]. Bidirectional checking also has two modes of operation: one where an expected type is known ("checking" mode); and another where an expected type is not known, and hence a type for the term being checked must be computed ("synthesizing" mode). These pairs of modes are orthogonal, and so incremental checking has four modes of operation: creating/checking, creating/synthesizing, non-creating/checking, and non-creating/synthesizing. Note that from the point of view of memory usage, the non-creating/checking mode is very nice, since incremental checking does not produce any output at all (neither a term nor its type) in that case.

2.1 Formalization

Incremental checking is formalized in Figure 1 using two judgments. The first,

$$\Gamma \,|\, I \;\Rightarrow^c\; t : T \mid I',$$

is intended to hold when in standard LF typing context Γ, a prefix of the textual input I is consumed to produce a term t of synthesized type T, with remaining suffix I' of the input I. If the boolean flag c is true, then we are in (term) creating mode, and the AST for the term t is actually constructed. If c is false, we are not, and though the term t is listed in the judgment, an implementation of incremental checking need not actually construct it. We model input as a list of tokens, thus abstracting away from lexical issues. We write x, I for the list of input tokens with variable x at the head, followed by I, and similarly for other tokens. The other tokens are λ, Π, *type*, *kind*, @ for application, and : for ascription. We rely here on a simple prefix syntax for the textual form of LF (without presenting its concrete syntax). We assume that variables are tacitly renamed when introduced into Γ to avoid having duplicate declarations for them there. We use a unified context Γ for both declared constants and bound variables. As standard, we assume contexts are well-formed in all rules, where we elide the obvious definition of well-formedness of contexts. The second judgment, $\Gamma \,|\, I \ \Leftarrow^c \ t : T \ | \ I'$ is similar, except we are here in checking mode, so T is an input to the judgment, rather than an output.

We write $\cong$ for standard LF definitional equality, and include it explicitly in the rules. Note that if terms use constants c defined to be λ-abstractions, it is not convenient to require that all input terms are in canonical form (since we may wish to write $(c\ a)$ in a term, instead of its β-short form). Hence, we take definitional equality to be as for standard LF (i.e., α-equivalence of β-short,η-long normal forms), and not α-equivalence, as used for canonical forms LF. For the same reason, we also include support for explicit ascriptions. Note that for incremental checking, it is most convenient to use ascriptions with the prefix syntax ": $T\ t$", where the type T comes before the term t to which it is ascribed. This is so that the term t can be consumed from the input in checking mode (checking against type T). Note that we can drop ascriptions after we have parsed them, so we do not need to have ascription as a term construct (for the terms created by incremental checking).

We incorporate our crucial observation about when we can avoid building ASTs of arguments in applications in the application rule, using the boolean flag of the judgment. If the variable x does not occur free in T_2, then we stipulate that the result of substituting a non-existent term t_2 for x in T_2 is just T_2. The reader might wonder when our boolean flag could ever be false. An implementation of incremental checking (or any LF checking algorithm, for that matter) should provide top-level commands for declaring and defining constants, as well as for checking the types of terms. In a command to check the type of a term, it is not necessary to create the term itself, and hence in that case, we can use the incremental checking judgment with the boolean flag initialized to false.

2.2 Correctness

We can easily establish correctness of incremental checking in two steps. First, we define an interpretation of our two judgments:

$$\begin{array}{lcl} \Gamma \,|\, I \ \Leftarrow^c \ t : T \ | \ I' & \mapsto & \Gamma \ \Leftarrow \ t : T \\ \Gamma \,|\, I \ \Rightarrow^c \ t : T \ | \ I' & \mapsto & \Gamma \ \Rightarrow \ t : T \end{array}$$

Our rules exactly match a standard bidirectional LF checking algorithm under this interpretation (dropping the ascription rule whose interpretation is obvi-

$$\frac{\Gamma \,|\, I \;\Rightarrow^c\; t : T' \mid I' \qquad T' \cong T}{\Gamma \,|\, I \;\Leftarrow^c\; t : T \mid I'}$$

$$\frac{}{\Gamma \,|\, \mathit{type}, I \;\Rightarrow^c\; \mathit{type} : \mathit{kind} \mid I}$$

$$\frac{\Gamma(x) = T}{\Gamma \,|\, x, I \;\Rightarrow^c\; x : T \mid I}$$

$$\frac{\Gamma \,|\, I \;\Rightarrow^c\; t_1 : \Pi x : T_1 .\, T_2 \mid I' \qquad \Gamma \,|\, I' \;\Leftarrow^{c \,\vee\, x \in FV(T_2)}\; t_2 : T_1 \mid I''}{\Gamma \,|\, @, I \;\Rightarrow^c\; (t_1\; t_2) : [t_2/x]T_2 \mid I''}$$

$$\frac{\Gamma, x : T_1 \,|\, I \;\Leftarrow^c\; t : T_2 \mid I'}{\Gamma \,|\, \lambda, x, I \;\Leftarrow^c\; \lambda x.\, t : \Pi x : T_1 .\, T_2 \mid I'}$$

$$\frac{\Gamma \,|\, I \;\Leftarrow^c\; T_1 : \mathit{type} \mid I' \qquad \Gamma, x : T_1 \,|\, I' \;\Rightarrow^c\; T_2 : \kappa \mid I'' \qquad \kappa \in \{\mathit{type}, \mathit{kind}\}}{\Gamma \,|\, \Pi, x, I \;\Rightarrow^c\; \Pi x : T_1 .\, T_2 : \kappa \mid I''}$$

$$\frac{\Gamma \,|\, I \;\Leftarrow^c\; T : \mathit{type} \mid I' \qquad \Gamma \,|\, I' \;\Leftarrow^c\; t : T \mid I''}{\Gamma \,|\; :, I \;\Rightarrow^c\; t : T \mid I''}$$

Fig. 1. Incremental checking rules for LF

ously admissible). The second issue is to verify that the outputs of judgments are well-defined. This could fail to be the case if a rule required creation of a term or type when one of its subterms had not been created or when a meta-theoretic operation used to create the expression was undefined. We can easily verify that whenever the boolean flag c in a judgment in the conclusion is true, then the subterms of the term t being created in that judgment all exist, and similarly for the type T. Of course, in the application case, as explained above, we avoid substituting a non-existent t_2 for x in T_2 in the case when $x \notin FV(T_2)$.

2.3 Empirical Results

Figures 2 and 3 compare a prototype incremental checker ("inc") against the Twelf checker, version 1.5R1; and also against checkers produced by signature compilation ("sc") [12,7]. Twelf is implemented in SML/NJ, while the other checkers are implemented in C++. Signature compilation takes an LF signature and generates source code for a proof checker specialized for checking terms expressed with respect to that signature. It is included here as the fastest LF checker of which the author is aware. Note that signature compilation is an orthogonal optimization to incremental checking, and could be applied to generate signature-specialized incremental checkers, with an additional expected speedup. This remains to be implemented, however. All times are reported as wallclock times in seconds (to two significant digits) and are the average of three runs on a lightly loaded Intel Core Duo CPU 1.20GHz, 2MB cache, 1.5MB main memory, running Linux 2.6.18. A timeout of 1800 seconds was imposed.

The benchmarks used in these examples are the same as considered in the work on signature compilation, which also compares emitted checkers with a few other high-performance checkers [12]. The EQ benchmarks are an artificial family of benchmarks using encoded congruence reasoning. The QBF benchmarks are generated from a simple QBF (Quantified Boolean Formulae) solver, written by the author, from some easy QBF benchmark formulas. The benchmark proofs all use a form of implicit LF, implemented in both the incremental checker and the specialized checkers emitted by signature compilation. This form uses explicit holes ("_") for omitted arguments, and requires that values for these holes be determined from the types of subsequent arguments (in the spine form of the application including the holes). This design allows holes to be filled in by higher-order matching (as opposed to unification) in the higher-order pattern fragment.

The figures show substantial improvements over the custom checker emitted by signature compilation, as well as a large advantage over Twelf. It should be noted that Twelf is not designed as a high performance checker, and has many powerful features beyond LF type checking. Twelf is included here as a checker not written or co-written by the author. Note that other theorem prover formats do not directly support HOAS, and so comparing with them would require an encoding of proofs, which is outside the scope of this evaluation. We do not consider full results for memory usage, but note that for the biggest benchmark (toilet_02_01.3), incremental checking requires peak memory usage of 4.1 MB to check the proof, which is less than the proof's size in ASCII text (8.2 MB). Timely and efficient reclamation of memory is achieved using manual increments and decrements of expression reference counts. Manual reference counting is extremely error-prone and tedious to debug. The excellent valgrind memory debugging tool provided invaluable assistance in tracking down memory leaks and errors.

3 LF with Side Conditions

As discussed above, we must be able to handle the resolution inference rule efficiently in our meta-logic for use with SMT solvers. These solvers inherit

benchmark	**size**	**inc**	**sc**	**Twelf**
gen100	20 KB	0.04	0.15	0.82
gen150	30 KB	0.05	0.29	1.6
gen200	40 KB	0.08	0.46	2.3
gen250	50 KB	0.12	0.67	3.3
gen300	60 KB	0.14	0.96	4.6
gen350	71 KB	0.18	1.2	5.9

Fig. 2. Checking times for EQ benchmarks

benchmark	**size**	**inc**	**sc**	**Twelf**
cnt01e	179 KB	0.25	0.28	4.0
tree-exa2-10	381 KB	0.35	0.50	6.1
cnt01re	267 KB	0.23	0.39	7.4
toilet_02_01.2	1.1 MB	0.92	1.3	150
1qbf-160cl.0	1.5 MB	0.98	1.1	750
tree-exa2-15	4.3 MB	3.7	5.8	timeout
toilet_02_01.3	8.2 MB	7.1	11.5	timeout

Fig. 3. Checking times for QBF benchmarks

the critical use of propositional resolution in clause learning from modern SAT solvers, and no successful meta-logic for SAT or SMT can fail to provide adequate support for resolution. As noted above, in pure LF, encoding the resolution rule is problematic, due to the need to enforce, via subproofs associated with each use of the rule, rather complex side conditions. The proposal here is to allow the signature designer to specify side conditions for encoded inference rules by means of programs written in a simple functional programming language. Explicit proofs of the side condition are not given when the encoded rule is applied. Rather, the type checker runs the side condition code and confirms that it succeeds producing the expected output. LF augmented with side conditions we call LFSC.

To support this, the syntax for Π-abstractions is augmented to allow them also to be of the form $\Pi x : \mathit{run}\ C\ t. T$. The domain type expresses the requirement that running code C should succeed producing output term t. For example, a resolution proof system may be encoded as in Figure 4 using side conditions. The figure uses Twelf syntax for readability, though the LFSC checker described below uses a prefix concrete syntax. The intention here is to use HOAS for encoding propositional variables. So the type `var` has no constructors. Clauses are lists of positive and negative occurrences of

```
var : type.

lit : type.
pos : var -> lit.
neg : var -> lit.

clause : type.
cln : clause.
clc : lit -> clause -> clause.

holds : clause -> type.
R : {c1:clause} {c2:clause} {c3:clause}
    (holds c1) ->
    (holds c2) ->
    {v:var}
    {r : run (resolve c1 c2 v) c3}
    (holds c3).
```

Fig. 4. Propositional resolution system using a side condition for `R`

variables. The resolution rule `R` states that if clauses `c1` and `c2` hold, then so does clause `c3` obtained by resolving `c1` and `c2` on variable `v`. For space reasons, the code implementing `resolve` must be omitted. Because all three clauses can be filled in from the types of subsequent arguments, all uses of `R` are of the following form, where `P1` and `P2` are the proofs that the clauses corresponding to `c1` and `c2` hold, and `v` is the variable upon which to resolve:

```
(R _ _ _ P1 P2 v)
```

No argument is given for the proof of the side condition, since the type checker verifies that it holds without any argument. One might wish to pursue an alternative design, where the type checker fills in a missing argument constituting a proof of the side condition. For example, as done in several of the works cited above, we could allow side conditions to be expressed using logic programs, and then take traces of runs of the logic programs as the proofs of the side conditions. This design would suffer from the problem that different runs of the logic program computing the same result would, in general, not be definitionally equal. Hence, the system would risk losing irrelevance of the computations used for establishing the side conditions. With no explicit proof term given, this problem is avoided.

We next give an informal description of the functional language proposed for use in LFSC, before turning to its syntax and informal operational semantics. The design of LFSC is not sensitive to the exact design of this language, which could easily be changed. We first note several high-level properties of this rather restricted language. It is a first-order, monomorphic, simply

$$
\begin{aligned}
C ::= {} & x \mid\mid c \mid\mid N \mid\mid (\odot\, C_1\ \cdots\ C_{n+1}) \mid\mid (c\, C_1\ \cdots\ C_{n+1}) \\
& \mid\mid (\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_{n+1}\ C_{n+1})) \mid\mid (\mathit{do}\ C_1\ \cdots\ C_{n+1}) \\
& \mid\mid (\mathit{let}\ x\ C_1\ C_2) \mid\mid (\mathit{markvar}\ C) \mid\mid (\mathit{ifmarked}\ C_1\ C_2\ C_3) \mid\mid (\mathit{fail}\ T)
\end{aligned}
$$

$$
P ::= (c\ x_1\ \cdots\ x_{n+1}) \mid\mid c
$$

Fig. 5. Syntax for code (C) and patterns (P).

typed functional programming language with pattern matching. Programs are mostly without mutable state, although there is a feature for marking LF variables. This feature enables efficient implementation of resolution. Additionally, code can fail, either explicitly using *fail*, or by failing to match a piece of inductively defined data against any of the patterns in a match expression. Programs may not operate on dependently typed data. Data must be at least weak head normalized (to remove LF λ-abstractions) before attempting to take a step of program evaluation. Programs are type checked, but are not statically checked for termination or coverage. There are also no facilities for proving properties about programs. The latter would generally require induction, which would begin to take us too far from the core ideas of LF. As part of the trusted computing base, programs must be trusted in any event. Finally, we currently do not support derived rules with side conditions. Side conditions may only be stipulated for declared (primitive) rules. This restriction fits well with the decision to avoid verification of side condition programs, since without induction to reason about program behavior, derived rules could be supported only with simple aggregation of side conditions of primitive rules.

3.1 Code Syntax

Figure 5 gives the syntax for code (C). We write $\odot$ for arithmetic operations. In multi-arity notation such as $(C_1\ \cdots\ C_{n+2})$, the number n is a natural number (including possibly 0); so for this example, at least two terms must be present in the list of terms $C_1, \ldots, C_{n+2}$. Evaluation is call-by-value, with earlier termination on failure.

Application. Expressions $(c\ C_1\ \cdots\ C_{n+1})$ are either of term constants or program constants to arguments. In the former case, the application is constructing a new piece of inductive data. In the latter, it is invoking a program.

Match. Expressions $(\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_{n+1}\ C_{n+1}))$ evaluate the scrutinee C to a piece of inductively defined data, and then seek to match that piece of data against one of the given simple patterns $P_1, \ldots, P_{n+1}$ in that order. Successfully matching against a pattern P_i binds the appropriate subdata to the variables in the pattern. The body C_i of the match is then evaluated and its result returned for the result of the match expression.

Do. Expressions (*do* $C_1 \ \cdots \ C_{n+1}$) evaluate each of $C_1, \ldots, C_{n+1}$ in turn, and return the value of the last. This is useful for checking that several conditions in a row do not fail.

Let. Expressions (*let* $x \ C_1 \ C_2$) are as standard in functional languages. The value (if any) of C_1 is substituted for x before evaluating C_2.

Markvar. The code (*markvar* C) first evaluates C. If the result is an LF variable, then this toggles a mark on that variable, and then returns the variable. These marks are useful in implementing resolution. We support marking just LF variables instead of marking arbitrary LF terms, because it is convenient to map all occurrences of the same (as determined by scoping) variable to the same in-memory representation. The same is not true for arbitrary terms, particularly in the presence of a non-trivial definitional equality (even that of just canonical forms LF, let alone that of the original LF): an indexing structure would be needed, imposing extra implementation complexity and more importantly, a runtime performance penalty.

Ifmarked. The code (*ifmarked* $C_1 \ C_2 \ C_3$) evaluates C_1. If the result is not a variable, evaluation fails. Otherwise, if the result is marked, we evaluate C_2; otherwise, we evaluate C_3.

Fail. We have (*fail* T) for explicitly indicating failure. The fail term is treated as having the given type T.

3.2 Checking Proofs from a SAT Solver

One of the author's students, Duckki Oe, has implemented a modern clause-learning SAT solver and instrumented it to produce proofs in LFSC format. Discussion of how this is done can be found in another paper, currently under review [9]. From this solver we obtain large resolution proofs expressed with respect to the LFSC signature given in Figure 4. These proofs begin with λ-abstractions for the propositional variables and also for the clauses initially assumed to hold. A refutation of these initial clauses then proceeds to deduce the empty clause by resolution. It is practically necessary to structure these proofs using lemmas, or they would otherwise explode in size. For incremental, online checking, we cannot wait until the final inference to determine which lemmas are relevant (and hence which must be checked). We are operating under the assumption that proofs may be too large to fit into main memory, and under that assumptions, we cannot defer checking proofs of lemmas. Lemmas can be supported with the following definition (in Twelf format), which uses HOAS to give a name to a proof of a first clause for use in the proof of a second:

```
satlem : {c1:clause}{c2:clause}
         (holds c1) -> ((holds c1) -> (holds c2)) ->
         (holds c2)
   = [c1][c2][u1][u2] (u2 u1).
```

A standard bidirectional type checking algorithm would require that we give the type for our large refutation proof, since that proof is a λ-abstraction. This type would be a Π-abstraction listing all the propositional clauses and assumptions of our initial clauses. Unfortunately, that Π-abstraction is too large to construct easily. The SAT benchmarks we are checking have sizes ranging from 1453 variables and 12531 clauses to 204664 variables and 609478 clauses. The latter would result in a Π-abstraction with nesting depth 814142. The C++ implementation of a prototype incremental LFSC checker (used also for the pure LF experiments above) compiles to assembly code allocating rather large stack frames of size 1404, which would thus require 1GB of stack (stack frame size times nesting depth) to parse and check. This seems to be prohibited on the author's test machine. Crafting the checker to be iterative instead of recursive would help control the memory usage, but we opt for a different approach.

We allow λ-abstractions to include the type for the bound variable if they occur in a synthesizing position. This simple idea is not sufficient to solve our problem, since it would generally still be necessary after parsing and type checking $\lambda x : A. M$ to restore any previous type declared for x in the surrounding context. The straightforward implementation of this would require saving that previous type on the stack, and thus preclude tail recursion. The solution is essentially to arrange matters so that we do not need to restore that previous type. The previous type does not need to be restored for a variable if the λ-abstraction occurs in a return position in the term: i.e., at a rightmost position in the term. We may simply keep track during (incremental) type checking of whether or not we are in such a position, and if so, we may make a tail call for checking the body of a λ-abstraction (with a type for the bound variable or not). Since the implementation uses a shared context for variables and constants, we allow this behavior only in a specially designated top-level "`check`" command, after which the type checker must exit (to avoid incorrectly reporting type A for x after completing checking of $\lambda x : A. M$).

The type checker cannot make tail calls when checking applications of the resolution rule `R`, since after checking the two subproofs, it must run the side condition code to compute the resolvent. But the type checker may make a tail call for the second subproof in a use of `satlem`. This will be a λ-abstraction, for whose body we may also make a tail call. Hence, if the resolution proof is structured as a right-nested sequence of uses of `satlem`, where the lemmas derived may use resolution unrestrictedly, we can use tail calls for the sequence of lemmas. This enables checking large proofs that would certainly exceed the allowed stack size without tail calls. Fortunately, structuring the proof as a sequence of lemmas is quite natural for clause-learning SAT solvers, since the

benchmark	proof size	check time
manol-pipe-g6bid	32 MB	237
manol-pipe-g7n	32 MB	397
velev-eng-uns-1.0-04	34 MB	7750
velev-sss-1.0-cl	4.8 MB	319
een-tipb-sr06-par1	35 MB	24.4
een-tipb-sr06-tc6b	8.6 MB	35.2
manol-pipe-c10ni_s	46 MB	79.4
manol-pipe-f6b	20 MB	1720
manol-pipe-f6n	20 MB	1810

Fig. 6. Proof checking times (in seconds) for large resolution proofs

lemmas recorded are just the clauses learned.

3.3 Empirical Results

Figure 6 presents proof checking times for an incremental LFSC checker on proofs produced by the SAT solver mentioned above. The benchmark formulas used to generate these proofs are easier formulas from SAT Race 2008 (making them still quite challenging to solve). We conjecture that variability in checking time may be due to different lengths of derived clauses. Profiling several large examples reveals that around 90% of the runtime of the proof checker is going into interpreting the side condition code.

4 Conclusion

Incremental checking and LF with Side Conditions (LFSC) hold the promise of bringing the flexibility and convenience of LF to SMT solvers. Proof checking time still lags behind the time to generate the proof from high-performance solvers. The next step to addressing this is to apply signature compilation to incremental LFSC checking [12]. This will enable specialization of type checking to the constants of the signature, as well as compiled side condition code. Future work also includes support for conversion to clausal form and theory reasoning.

References

[1] A. Appel. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science*, 2001.

[2] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05), Post-Proceedings*, 2006.

[3] L. Bauer, S. Garriss, J. McCune, M. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, September 2005.

[4] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2006.

[5] G. Necula and S. Rahul. Oracle-Based Checking of Untrusted Software. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, 2001.

[6] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[7] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[8] 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.

[9] 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. under review.

[10] 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.

[11] D. Wu, A. Appel, and A. Stump. Foundational Proof Checkers with Small Witnesses. In D. Miller, editor, *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 264–274, 2003.

[12] 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.

[13] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *International Conference on Computer Aided Design*, pages 279–285, 2001.