Towards an SMT Proof Format

Aaron Stump and Duckki Oe
Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA

Abstract

The Edinburgh Logical Framework (LF) extended to support side condition code (LFSC) is advocated as a foundation for a proof format for SMT. The flexibility of the framework is demonstrated by example encoded inference rules, notably propositional resolution. Preliminary empirical results obtained with a SAT solver producing proofs in LFSC format are presented.

1 Introduction

Given the complexity of modern SMT solvers, practical methods for verifying either the solvers or their results are desirable. A well-known method for verifying reports of unsatisfiability is for solvers to produce a proof refuting the formula. Just as was done by the SMT-LIB initiative for SMT input formulas, it is highly desirable to develop a common format for such proofs, which different solvers could target. Since SMT solvers employ a variety of reasoning methods, it is at the least premature to standardize around a particular set of axioms and inference rules. Instead, this paper proposes to standardize around a logical framework, in which proof systems corresponding to these different reasoning methods can be encoded. This framework is called LF with Side Conditions (LFSC), and is an extension of the Edinburgh Logical Framework (LF), which has been used successfully for representing proofs in applications like proof-carrying code [3, 1]. LF allows proof systems to be encoded succinctly in a type theory with special support for variable-binding constructs, which occur frequently in both formula and proof syntaxes. LFSC extends LF with better support for rules with computational side conditions. Encoded inference rules may require certain side conditions to succeed for every application of the rule, where the side conditions are described using a simple functional programming language. This paper introduces the syntax and informal semantics of LFSC (Section 2), and shows how LFSC can be used to encode resolution inferences (Section 3). Empirical results using a solver called CLSAT are presented (Section 4).

$(\textit{declare}\ c\ T)\quad ||\quad (\textit{define}\ c\ t)\quad ||\quad (\textit{opaque}\ c\ t)\quad ||\quad (\textit{check}\ t)\ ||$
$(\textit{program}\ c\ ((x_1\ t_1)\ \cdots\ (x_{n+1}\ t_{n+1}))\ t\ C)$

Figure 1: Top-level commands.

2 LF with Side Conditions

This section presents the syntax and gives an informal description of the typing and operational semantics for LF with Side Conditions (LFSC). Commands are described first. Then terms (t), types (T), and side condition code (C).

2.1 Command Syntax

A proof is given as a list of commands, listed in Figure 1 and described next:

Declarations. Constants c may be declared (with *declare*) to have type T, or may be defined (with *define*) to equal t. They may also be given an opaque definition (with *opaque*), in which case subsequent type checking will use only the fact that c has type T computed for t, and not the fact that c equals t. Declared constants are used in the LF encoding methodology to represent primitive symbols of an object-language. For SMT, these include theory functions and predicate symbols. They are also used to represent axioms and inference rules. Opaque definitions (*opaque* c t) are introduced in LFSC to allow the memory required for the defining term t to be reclaimed after checking its type. Ordinary definitions (introduced with *define*) must retain the defining term, in case the subsequent proof depends on the fact that c equals t.

Checks. We can check a term t by computing a type for it.

Programs. Singly recursive programs may be defined at the top-level of the input using *program* commands (adding support for mutual recursion would not be difficult). The recursive program is called c; its input variables are $x_1, \ldots, x_{n+1}$, with types $t_1, \ldots, t_{n+1}$; its return type is t, and its body is C.

One simple usage scenario for these commands is the following. Following standard LF terminology, a collection of declarations and definitions is called a *signature*. The SMT-LIB organization may provide a signature specifying SMT input syntax for various logics. It is also desirable to provide an example signature for axioms and inference rules for that syntax, perhaps based on the one in progress by the authors. SMT solver implementors may use this example signature, or write their own to specify their axioms and inference rules. Untrusted proofs are then not to use declarations, since these may non-conservatively extend the logic; but only definitions and *check* commands.

2.2 Term and Type Syntax

Figure 2 gives the syntax for LFSC terms t and types T. In multi-arity notation such as $(t_1\ \cdots\ t_{n+2})$, the number n is a natural number (including possibly

$$\begin{array}{rcl} t & ::= & x \mid\mid c \mid\mid \_ \mid\mid N \mid\mid (t_1\ \cdots\ t_{n+2}) \mid\mid (\backslash\ x\ t) \mid\mid (\$\ x\ T\ t) \mid\mid (:\ T\ t) \\ \\ T & ::= & c \mid\mid (T\ t_1\ \cdots\ t_{n+1}) \mid\mid (!\ x\ r\ T) \\ \\ r & ::= & (\hat{}\ C\ t) \mid\mid T \end{array}$$

Figure 2: Syntax for terms (t) and types (T).

0); so for this example, at least two terms must be present in the list of terms $t_1, \ldots, t_{n+2}$. We briefly describe the various constructs, and how they are used in representing theories.

Variables and constants. We write x for variables, c for constants, and N for unbounded integer literals (adding support also for rational literals would be straightforward). The following special constants c are assumed present: *type*, which is the type for types; and *mpz*, the type for integers.

Applications. Term-level applications $(t_1\ \cdots\ t_{n+2})$ consist of a functional term t_1 and its arguments $t_2\ \cdots\ t_{n+2}$. Applications are used to represent applications of SMT function and predicate symbols to arguments. They are also used to represent applications of logical connectives to formulas. Finally, they are also used to represent applications of inference rules and axioms to arguments for their quantified variables and hypotheses.

Holes. Applications may use holes $\_$, whose value is to be filled in from the types of subsequent arguments.

Indexed types. Type-level applications $(T\ t_1\ \cdots\ t_{n+1})$ are used for indexed types. In a standard functional language, we can easily define a datatype `Pf` for proofs. But type checking cannot enforce proof checking, because standard type systems do not have a way to express that a proof is a proof of a given formula. Thus, we could apply an encoded inference rule like modus ponens to any two terms of type `Pf`, regardless of whether or not the first is a proof of $\phi \to \psi$ and the second is a proof of ϕ. With indexed types, the type system of LF (and LFSC) is able to track the formula proved by a particular proof, as an index to the type. So an encoded proof of ϕ would have type "(`Pf F`)", assuming `F` is the encoding of formula ϕ. Using indexed types, axioms and inference rules can be encoded as term constructors in such a way that they cannot be applied in cases where the object-language inference would be incorrect. We can, for example, declare (encoded) modus ponens to be a term constructor accepting one argument of type "`(Pf (imp F1 F2))`", and another argument of type "`(Pf F1)`", and constructing a resulting term of type "`(Pf F2)`".

Lambda abstractions. The expressions $(\backslash\ x\ t)$ are conventionally written in mathematical notation $\lambda x.\ t$. They are anonymous functions accepting arguments for input variable x, and returning t. In the LF encoding methodology, lambda abstractions are used to encode object-language binding constructs.

$$
\begin{array}{rcl}
C & ::= & x \mid\mid c \mid\mid N \mid\mid (\odot\ C_1\ \cdots\ C_{n+1}) \mid\mid (c\ C_1\ \cdots\ C_{n+1}) \\
 & & \mid\mid (\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_{n+1}\ C_{n+1})) \mid\mid (\mathit{do}\ C_1\ \cdots\ C_{n+1}) \\
 & & \mid\mid (\mathit{let}\ x\ C_1\ C_2) \mid\mid (\mathit{markvar}\ C) \mid\mid (\mathit{ifmarked}\ C_1\ C_2\ C_3) \mid\mid (\mathit{fail}\ T) \\
 & & \\
P & ::= & (c\ x_1\ \cdots\ x_{n+1}) \mid\mid c
\end{array}
$$

Figure 3: Syntax for code (C) and patterns (P).

The variable bound by an object language binder is represented by a λ-bound variable in LF. It is customary with LF to omit the type for the variable x, because the LF type checking algorithm will fill it in from the context surrounding the lambda abstraction. It turns out that for representing large proofs, it can be necessary to allow the type for the bound variable to be specified in the λ-abstraction. In this case, we write (\$ x T t) for $(\lambda\, x : T.\ t)$.

Pi abstractions. The expressions (! x t_1 t_2) are conventionally written in mathematical notation $\Pi\, x : t_1.\ t_2$. These are the types for functions accepting inputs x of type t_1 and producing outputs of type t_2, where t_2 may contain x free. Such types are called *dependent function types*, because the output type can depend on the input argument. A programming example may help provide intuition for such types. Imagine a function that accepts a natural number n and returns an array of *int*s of length n. Such a function might be given dependent function type $\Pi\, x : \mathit{nat}.\ \mathit{int}[n]$ (writing $\mathit{int}[n]$ for the type of arrays of *int*s of length n). Such dependencies are used heavily when giving the types for encoded inference rules, since we wish to capture dependencies such as described above between the indices of the types of subproofs of modus ponens. As its crucial addition to standard LF, LFSC allows the domain type of an abstraction to be of the form (^ C t). The intuitive meaning of this expression is that running code C should produce result t. Note that t may be a hole, in which case it will be filled in with the output produced by running C.

Ascriptions. Expressions (: T t) state that T is the type for term t. Type checking an ascription requires verifying that this is indeed the case. Ascriptions are needed just in giving definitions.

2.3 Code Syntax

Figure 3 gives the syntax for code (C). We write $\odot$ for arithmetic operations on unbounded integers. Code constructs are used for writing strongly typed first-order, monomorphic functional programs. First-order means that unlike well-known functional languages such as HASKELL and OCAML, functions may not be passed as arguments to programs or produced as results. Monomorphic means that, again unlike those languages, programs cannot be defined polymorphically, to operate uniformly on all types of data. The programs are mostly without mutable state, although there is a feature for marking LF variables, demonstrated below. The resulting language supports programs where inductively defined data may be recursively decomposed (using `match`) and constructed. 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.

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 C to a piece of inductively defined data, and then seek to match that piece of data against one of the given simple patterns $P_1, \ldots, P_{n+1}$. 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 $(\mathit{do}\ C_1\ \cdots\ C_{n+1})$ 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 (it is not related to `do` in HASKELL).

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

Markvar. The code $(\mathit{markvar}\ C)$ 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 the important example of resolution inferences below (Section 3).

Ifmarked. The code $(\mathit{ifmarked}\ C_1\ C_2\ C_3)$ 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 $(\mathit{fail}\ T)$ for explicitly indicating failure. The fail term is treated as having the given type T.

3 Encoding Resolution Proofs

This section presents an important example of encoding a proof system in LFSC, namely propositional resolution. Resolution is used during clause learning in state-of-the-art SAT and SMT solvers [5]. A proof format proposed in 2005 by van Gelder for SAT solvers is based directly on resolution. Encoding a resolution proof system in LFSC is thus a critical step in encoding proof systems for the more general logics of SMT.

3.1 Encoding Clauses

The first step in encoding any proof system in LFSC (or pure LF) is to encode the formulas of the logic. Once these are encoded, we can consider how to encode proofs. In the case of propositional resolution, the formulas of the logic are propositional clauses. Figure 4 lists LFSC declarations for these. We first

```
(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)))
```

Figure 4: Clauses

declare an LFSC type `var`, for propositional variables. We do not give any term constructors for variables, for reasons which will be given shortly. Next, we declare a type `lit` for propositional literals, with two constructors, `pos` and `neg`. We will use these for positive and negative occurrences, respectively, of a variable in a clause. Note that the type given for both `pos` and `neg`, namely

```
(! x var lit)
```

says that `pos` and `neg` are functions taking input `x`, which is a variable, and producing a literal. Finally, the type `clause` of clauses is declared with constructors `cln` for the empty clause, and `clc` for cons'ing a literal (`x`) onto the front of a clause (`c`). Assuming that P and Q are propositional variables, then a clause like $P \vee \neg Q$ is encoded with these declarations as

```
(clc (pos P) (clc (neg Q) cln))
```

3.2 Encoding Resolution Inferences

Figure 5 gives three declarations for encoding binary resolution with factoring. More complex forms of resolution such as hyperresolution should also be encodable, but this is future work. First, we declare an indexed type `holds`, with the intention that "`(holds C)`" is the type for proofs that clause `C` holds. We defer consideration of `resolve`. The declaration of `R`, which encodes the resolution inference rule, states that `R` takes clauses `c1`, `c2`, and `c3` as inputs; then a proof that clause `c1` holds (this is the input argument `u1` of type "`(holds c1)`"), and a proof that clause `c2` holds; and a variable `v`. If running the code "`(resolve c1 c2 v)`" produces clause `c3`, then `R` constructs a term of type "`(holds c3)`", representing a resolution inference deriving `c3`. Since the values of the inputs `c1`, `c2`, and `c3` to `R` can all be filled in from subsequent arguments (in particular, `u1` to fill in `c1`, `u2` to fill in `c2`, and `r` to fill in `c3`), we can use holes "`_`" for these inputs whenever `R` is used. This is demonstrated in Section 3.3 below.

The side condition to check for a resolution inference is as follows. If we are resolving clauses `c1` and `c2` on variable `v`, `v` must occur positively in `c1` and

```
(declare holds (! c clause type))

(program resolve ((c1 clause) (c2 clause) (v var)) clause
  (let pl (pos v)
  (let nl (neg v)
  (do (in pl c1)
      (in nl c2)
      (let d (append (remove pl c1) (remove nl c2))
         (dropdups d))))))

(declare R (! c1 clause (! c2 clause (! c3 clause
           (! u1 (holds c1)
           (! u2 (holds c2)
           (! v var
           (! r (^ (resolve c1 c2 v) c3)
            (holds c3)))))))))
```

Figure 5: Propositional resolution in LFSC

negatively in `c2`. All positive occurrences are removed from `c1` and all negative ones from `c2`. The resulting clauses are concatenated to produce the resolvent `c3`. Additionally, it is desirable to drop duplicate literals from the resolvent.

LFSC code to compute a resolvent from `c1`, `c2`, and `v` is given in the program `resolve` of Figure 5. This program relies on several helper programs, mostly relegated to the Appendix. Reading through the body of `resolve`, we can see that it first `let`-defines `pl` to be the positive literal for variable `v`, and `nl` to be the negative literal. It then checks that this positive literal is in `c1`, and the negative one in `c2`. It then `let`-defines `d` to be the result of appending the results of removing the positive literal from `c1` and the negative literal from `c2`. Finally, we return the result of dropping duplicates from `d`.

3.3 An Example Resolution Proof

For a very simple example, suppose our clause database consists of the following clauses: $\neg V_1 \vee V_2$, then $\neg V_2 \vee V_3$, then $\neg V_3 \vee \neg V_2$, and $V_1 \vee V_2$. A resolution derivation of the empty clause from these clauses is given in Figure 6. The encoding in LFSC of this proof is given in Figure 7. We can see in the last line of Figure 7 three applications of `R`, corresponding to the three resolution inferences in Figure 6. As mentioned above, the clauses being resolved and the resolvent (the first three arguments to `R`) do not need to be mentioned when `R` is applied, because those clauses can all be filled in from subsequent arguments. That is why each use of `R` begins with three holes. The nested resolutions in Figure 6 are mirrored by the nested applications of `R` in Figure 7. Where Figure 6 lists the n'th clause (in the order the clauses were listed above) from our clause database, at the leaves of the proof tree; in those places Figure 7 lists

$$\dfrac{\dfrac{V_1 \lor V_2 \quad \neg V_1 \lor V_2}{V_2} \qquad \dfrac{\neg V_2 \lor V_3 \quad \neg V_3 \lor \neg V_2}{\neg V_2}}{\mathit{empty}}$$

Figure 6: An example refutation

```
(check ($ v1 var ($ v2 var ($ v3 var
       ($ x0 (holds (clc (neg v1) (clc (pos v2) cln)))
       ($ x1 (holds (clc (neg v2) (clc (pos v3) cln)))
       ($ x2 (holds (clc (neg v3) (clc (neg v2) cln)))
       ($ x3 (holds (clc (pos v1) (clc (pos v2) cln)))
       (R _ _ _ (R _ _ _ x3 x0 v1) (R _ _ _ x1 x2 v3) v2)))))))
```

Figure 7: LFSC encoding of the example refutation

`xn` (e.g. `x3` and `x0` in the leftmost innermost resolution).

Let us now consider the LFSC command in Figure 7 more carefully. This is a `check` command. The proof begins by λ-abstracting (with \$) all the propositional variable and assumptions that all the initial clauses hold. The use of λ-abstraction here comes from standard LF encoding methodology, where one seeks to represent object-language variables via LF variables. This confers one of the main advantages of using LF, namely that the encoding need not explicitly describe safe renaming of or substitution for bound variables. These features are provided by LF directly for λ-bound variables, and hence are inherited for free by any encoding that represents object-language variables by LF variables. The main benefit of using LF variables for propositional variables is that we can efficiently test equality of variables in LFSC code using variable marking. This is necessary when computing the resolvent: for example, when testing whether or not the negative literal for variable `v` occurs in the second clause being resolved (see the Appendix).

4 Preliminary Empirical Results

CLSAT is a SMT solver currently supporting the QF_IDL logic. It can solve SAT benchmarks in DIMACS format and SMT benchmarks in SMT-LIB format, generating resolution proofs for unsatisfiable SAT benchmarks. Proofs of SMT benchmarks are not yet supported. The use of resolution in deriving conflict clauses from conflicting clauses is well known [5]. Learned clauses are recorded as lemmas to keep the proof size from exploding (see also Section A). A refutation of the formula is then just a sequence of lemmas, culminating in the empty clause.

The benchmarks used are from SAT Race 2008 Test Set 1, which represent modern SAT benchmarks. Since these are quite large and difficult to solve, 10 easier ones out of the 31 unsatisfiable benchmarks in the set were selected (see

benchmark	pf (s)	overhd	sz (MB)	num R	check (s)	tot overhd
E-sr06-par1	4.56	196.10%	35	14316	14.75	11.54
E-sr06-tc6b	0.96	152.63%	8.4	8708	11.68	32.26
M-c10ni_s	6.62	34.01%	43	4578	10.90	2.55
M-c6nid_s	15.58	12.82%	33	72930	48.35	3.63
M-f6b	20.76	29.51%	30	1018638	3237.22	202.24
M-f6n	16.59	35.76%	26	847567	2848.03	233.42
M-g6bid	20.05	28.61%	27	797530	1165.57	75.05
M-g7n	16.12	43.03%	28	1006820	1707.43	151.93
V-eng-uns-1.0-04	25.04	29.27%	41	1692714	5913.22	305.57
V-sss-1.0-cl	4.18	46.15%	9.8	416200	553.30	193.92

Figure 8: Proof sizes and proof checking times

also Section D). Checking is carried out using a prototype LFSC checker [4]. The checker does not yet compile side condition code. Profiling reveals that around 90% of checking time is currently devoted to interpreting side condition code. This at least partially explains the much greater times required for checking proofs over producing them. The checker implements an optimization called incremental checking, where parsing and proof checking are interleaved. Abstract syntax trees for subterms of proofs are not constructed in memory at all, unless they are used in the theorem proved by a subproof.

The proofs of all derived lemmas are emitted by the solver. Excluding unnecessary lemmas is expected often to result in smaller proofs, but this requires storing all proofs until the end of the run. Incremental checking alone is currently not fast enough to keep up with the solver's proof production. But if this can be achieved, then the checker can consume proofs of unnecessary lemmas as they are produced. This would allow the solver to avoid storing them during the attempted refutation, and hence would be preferable, at least for proof checking purposes, to emitting just the needed lemmas.

Figure 8 shows results related to proof production and checking. The "pf" column gives the time to solve the benchmark and produce a proof (including time to write the proof to disk). The "overhd" column gives the percentage running time overhead incurred for proof production. The "sz" column gives the size of the generated proofs in megabytes. The "num R" column gives the number of resolutions. The "check" column gives proof checking time in seconds. The "totl overhd" column gives the ratio of proof production and checking time, to time needed to solve the benchmark without producing a proof. All experiments were performed on an Intel Core 2 Duo 2GHz, 4MB L2 Cache, 2GB memory, running MacOS 10.5.2.

5 Related Work and Conclusion

A recent paper by M. Moskal gives another approach to flexible proof checking, currently achieving much faster proof checking than reported here [2]. Moskal's approach uses term rewriting with a fixed (non-standard) reduction strategy to rewrite proofs to the theorems, if any, they prove. This formalism combines symbolic and functional programming, the former because the reduction strategy includes reduction beneath λ-abstractions. This will prove a hinderance to compilation. In contrast, LFSC separates a standard, naturally compilable programming language from more declarative aspects of proofs (particularly those involving bound variables). A further difference is that Moskal's formalism is untyped, while LFSC's enjoys the well-known benefits of strong typing. Finally, Moskal's formalism includes ad hoc features to support sound skolemization. While conversion to CNF has not yet been implemented in CLSAT, LF's direct support for higher-order abstract syntax enables introduction of new symbols without additional ad hoc features.

This paper has taken important first steps towards a meta-logical approach to SMT proofs, based on LF with Side Conditions (LFSC). The important case of propositional resolution has been evaluated with the CLSAT SAT solver, which produces LFSC proofs for unsatisfiable benchmarks. This work's achievement is in supporting large propositional resolution proofs with a general meta-logic. The use of a meta-logic paves the way for flexibly supporting theory inferences and CNF conversion. Future work includes proof production for full SMT reasoning in CLSAT, and compilation of side condition code for faster LFSC proof checking.

Acknowledgments. This work was partially supported by NSF award 0551697. Thanks are due to the SMT 2008 reviewers for helpful comments that improved the paper. Thanks also to Clark Barrett, Leonardo de Moura, Pascal Fontaine, and Cesare Tinelli for helpful discussions on LF-based proof checking for SMT.

References

[1] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[2] 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.

[3] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[4] 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.

[5] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Proceedings of 8th International Conference on Computer Aided Deduction (CADE 2002)*, 2002.

A Propositional Lemmas

To keep proofs as compact as possible, it is critical for encoded resolution proofs to be able to introduce lemmas for learned clauses. This is done with `satlem`, defined in Figure 9. Note the use of an ascription in giving this definition. If `P1` proves that clause `c1` holds, and if `P2` is a proof of clause `c2` from a hypothesis named x that `c1` holds, then "`(satlem _ _ P1 ( x P2))`" derives `c2` without hypotheses. In practice, the name of the hypothesis x can be determined by the clause number introduced by the SAT solver for the new clause.

B Helper Code for Resolution

The helper code called by the side condition program `resolve` of the encoded resolution rule `R` is given in Figures 10 and 11. We can note the frequent uses of `match`, for decomposing or testing the form of data. The program `eqvar` of Figure 10 uses variable marking to test for equality of LF variables. The code assumes a datatype of booleans `tt` and `ff`. It marks the first variable, and then tests if the second variable is marked. Assuming all variables are unmarked except during operations such as this, the second variable will be marked iff it happens to be the first variable. The mark is then cleared (recall that `markvar` toggles marks), and the appropriate boolean result returned. Marks are also used by `dropdups` to drop duplicate literals from the resolvent.

C Encoding Theory Reasoning

For SMT, it is, of course, necessary to add theory reasoning to our encoded propositional resolution calculus. The authors have not yet implemented this in CLSAT, and so these ideas must be considered preliminary. Let us distinguish sorted terms and formulas (a more unified view, as under discussion for the next revision of the SMT input syntax, should also be possible to support). If we wish, we can easily embed SMT type checking into LF type checking by using indexed types "`(term s)`" as the LF type for encodings of SMT terms of (encoded) sort `s`. Here we give an example theory inference rule, from Integer Difference Logic. This rule says that if $\neg\ x - y \ \leq\ c$ holds, then so does $y - x \leq d$, where c is an integer constant and d is the integer predecessor of c. The encoding of this inference, given in Figure 12, uses side condition code to compute the integer predecessor of c. The operations `mpz_add` and `mpz_neg` are integer operations implemented by the prototype LFSC checker. Using this rule, if `P` proves (the encoding of) $x - y \leq c$, then `(not<=<= _ _ _ _ P)` proves $y - x \leq d$, where $d = -c - 1$.

```
(define satlem (: (! c1 clause
                  (! c2 clause
                  (! u1 (holds c1)
                  (! u2 (! x (holds c1) (holds c2))
                     (holds c2)))))
               (\ c1 (\ c2 (\ u1 (\ u2 (u2 u1)))))))
```

Figure 9: Definition for propositional lemmas

```
(program eqvar ((v1 var) (v2 var)) bool
    (do (markvar v1)
        (let s (ifmarked v2 tt ff)
           (do (markvar v1) s))))

(program litvar ((l lit)) var
  (match l ((pos x) x) ((neg x) x)))

(program eqlit ((l1 lit) (l2 lit)) bool
  (match l1 ((pos v1) (match l2 ((pos v2) (eqvar v1 v2))
                                ((neg v2) ff)))
            ((neg v1) (match l2 ((pos v2) ff)
                                ((neg v2) (eqvar v1 v2))))))
```

Figure 10: Variable and literal comparison

```
(declare Ok type)
(declare ok Ok)

(program in ((l lit) (c clause)) Ok
  (match c ((clc l' c') (match (eqlit l l') (tt ok) (ff (in l c'))))
           (cln (fail Ok))))

(program remove ((l lit) (c clause)) clause
  (match c (cln cln)
           ((clc l' c' )
              (let u (remove l c')
                 (match (eqlit l l') (tt u) (ff (clc l' u)))))))

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

(program dropdups ((c1 clause)) clause
  (match c1 (cln cln)
           ((clc l c1')
            (let v (litvar l)
               (ifmarked v
                  (dropdups c1')
                  (do (markvar v)
                      (let r (clc l (dropdups c1'))
                         (do (markvar v) ; clear the mark
                             r))))))))
```

Figure 11: Operations on clauses

```
(declare not<=<=
  (! x (term Int) (! y (term Int) (! c mpz (! d mpz
  (! u (th_holds (not (<= (- x y) (an_int c))))
  (! r (^ (mpz_add ( mpz_neg c) (~ 1)) d)
    (th_holds (<= (- y x) (an_int d))))))))))
```

Figure 12: Example theory rule

benchmark	size (MB)	CLSAT	MINISAT	TINISAT
E-sr06-par1	8.4	1.54	1.46	1.43
E-sr06-tc6b	1.9	0.38	0.22	0.34
M-c10ni_s	10	4.94	43.42	7.14
M-c6nid_s	7.4	13.81	162.01	93.56
M-f6b	1.7	16.03	4.02	5.41
M-f6n	1.7	12.22	4.57	6.58
M-g6bid	1.8	15.59	3.60	3.99
M-g7n	1.1	11.27	2.75	6.46
V-uns-1.0-04	1.0	19.37	5.19	5.63
V-1.0-cl	0.18	2.86	0.41	0.21

Figure 13: Comparison of CLSAT with other solvers

D Solver Performance

CLSAT has its own lemma-learning SAT solver implementing watched literals, non-chronological backtracking, and conflict clause simplification. The SAT engine of CLSAT is primarily implemented by the second author, with other parts of CLSAT written by Timothy Simpson and Terry Tidwell. Figure 13 compares the running times of CLSAT, with proof production turned off, with those of MINISAT 2.0 beta and TINISAT 0.22. This figure shows that CLSAT is not too much slower than these two well-known fast modern solvers, and hence provides a reasonable basis for studying proof production.