Noname manuscript No.
(will be inserted by the editor)

SMT Proof Checking Using a Logical Framework

Aaron Stump · Duckki Oe · Andrew Reynolds · Liana Hadarean · Cesare Tinelli

Received: date / Accepted: date

Abstract Producing and checking proofs from SMT solvers is currently the most feasible method for achieving high confidence in the correctness of solver results. The diversity of solvers and relative complexity of SMT over, say, SAT means that flexibility, as well as performance, is a critical characteristic of a proof-checking solution for SMT. This paper describes such a solution, based on a Logical Framework with Side Conditions (LFSC). We describe the framework and show how it can be applied for flexible proof production and checking for two different SMT solvers, CLSAT and CVC3. We also report empirical results showing good performance relative to solver execution time.

Keywords Satisfiability Modulo Theories · Proof Checking · Edinburgh Logical Framework, LFSC

1 Introduction

Solvers for Satisfiability Modulo Theories (SMT) are currently at the heart of several formal method tools such as extended static checkers [16,3], bounded and unbounded model-checkers [1,21,14], symbolic execution tools [25], and program verification environments [10,49]. The main functionality of an SMT solver is to determine if a given input formula is satisfiable in the logic it supports—typically some fragment of first-order logic with certain built-in theories such as integer or real arithmetic, the theory of arrays and so on. Most SMT solvers combine advanced general-purpose propositional reasoning (SAT) engines with sophisticated implementations of special-purpose reasoning algorithms for built-in theories. They are rather complex and large tools, with codebases often between 50k

This work was partially supported by funding from the US National Science Foundation, under awards 0914877 and 0914956.

A. Stump, D. Oe, A. Reynolds, C. Tinelli
E-mail: {aaron-stump,duckki-oe,andrew-reynolds,cesare-tinelli}@uiowa.edu

L. Hadarean
E-mail: lsh271@nyu.edu

and 100k lines of C++. As a consequence, the correctness of their results is a long-standing concern. In an era of tour-de-force verifications of complex systems [24, 27], noteworthy efforts have been made to apply formal verification techniques to *algorithms* for SAT and SMT [29,18]. Verifying actual solver code is, however, still extremely challenging [28] due to the complexity and size of modern SMT solvers.

One approach to addressing this problem is for SMT solvers to emit independently checkable evidence of the results they report. For formulas reported as unsatisfiable, this evidence takes the form of a refutation proof. Since the relevant proof checking algorithms and implementations are much simpler than those for SMT solving, checking a proof provides a more trustworthy confirmation of the solver's result. Interactive theorem proving can benefit as well from proof producing SMT solvers, as shown recently in a number of works [12,11]. Complex subgoals for the interactive prover can be discharged by the SMT solver, and the proof it returns can be checked or reconstructed to confirm the result without trusting the solver. Certain advanced applications also strongly depend on proof production. For example, in the Fine system, Chen *et al.* translate proofs produced by the Z3 solver [32] into their internal proof language, to support certified compilation of richly typed source programs [13]. For Fine, even a completely verified SMT solver would not be enough since the proofs themselves are actually needed by the compiler. Besides Z3 other examples of proof-producing SMT solvers are CLSAT, CVC3, Fx7, and veriT [38,6,31,15].

A significant enabler for the success of SMT has been the SMT-LIB standard input language [5], which is supported by most SMT solvers. So far, no standard proof format has emerged. This is, however, no accident. Because of the ever increasing number of logical theories supported by SMT solvers, the variety of deductive systems used to describe the various solving algorithms, and the relatively young age of the SMT field, designing a single set of axioms and inference rules that would be a good target for all solvers does not appear to be practically feasible. We believe that a more viable route to a common standard is the adoption of a common *meta*-logic in which SMT solver implementors can describe the particular proof systems used by their solver. With this approach, solvers need to agree just on the meta-language used to describe their proof systems. The adoption of a sufficiently rich meta-logic prevents a proliferation of individual proof languages, while allowing for a variety of proof systems. Also, a single meta-level tool suffices to check proofs efficiently on any proof system described in the meta-language. A fast proof checker can be implemented once and for all for the meta-language, using previously developed optimizations [43,50,44, e.g.]. Also, different proof systems can be more easily compared, since they are expressed in the same meta-level syntax. This may help identify in the future a common core proof system for some significant class of SMT logics and solvers.

In this paper we propose and describe a meta-logic, called LFSC, for "Logical Framework with Side Conditions", which we have developed explicitly with the goal of supporting the description of several proof systems for SMT, and enabling the implementation of very fast proof checkers. In LFSC, solver implementors can describe their proof rules using a compact declarative notation which also allows the use of computational side conditions. These conditions, expressed in a small functional programming language, enable some parts of a proof to be established by computation. The flexibility of LFSC facilitates the design of proof systems that reflect closely the sort of high-performance inferences made by SMT solvers.

The side conditions feature offers a continuum of possible LFSC encodings of proof systems, from completely declarative at one extreme, using rules with no side conditions, to completely computational at the other, using a single rule with a huge side condition. We argue that supporting this continuum is a major strength of LFSC. Solver implementors have the freedom to choose the amount of computational inference when devising proof systems for their solver. This freedom cannot be abused since any decision is explicitly recorded in the LFSC formalization and becomes part of the proof system's trusted computing base. Moreover, the ability to create with a relatively small effort different LFSC proof systems for the same solver provides an additional level of trust even for proof systems with a substantial computational component—since at least during the developing phase one could also produce proofs in a more declarative, if less efficient, proof system.

We have put considerable effort in developing a full blown, highly efficient proof checker for LFSC proofs. Instead of developing a dedicated LFSC checker, one could imagine embedding LFSC in declarative languages such as Maude or Haskell. While the advantages of prototyping symbolic tools in these languages are well known, in our experience their performance lags too far behind carefully engineered imperative code for high-performance proof checking. This is especially the case for the sort of proofs generated by SMT solvers which can easily reach sizes measured in megabytes or even gigabytes. Based on previous experimental results by others, a similar argument could be made against the use of interactive theorem provers (such as Isabelle [37] or Coq [8]), which have a very small trusted core, for SMT-proof checking. By allowing the use of computational side conditions and relying on a dedicated proof checker, our solution seeks to strike a pragmatic compromise between trustworthiness and efficiency.

We introduce the LFSC language in Section 2 and then describe it formally in Section 3. In Section 4 we provide an overview of how one can encode in LFSC a variety of proof systems that closely reflect the sort of reasoning performed by modern SMT solvers. Building on the general approaches presented in this section, we then focus on a couple of SMT theories in Section 5, with the goal of giving a sense of how to use LFSC's features to produce compact proofs for theory-specific lemmas. We then present empirical results for our LFSC proof checker which show good performance relative to solver-execution times (Section 6). Finally, Section 7 touches upon a new, more advanced application of LFSC: the generation of certified interpolants from proofs.

The present paper builds on previously published workshop papers [45, 38–40] but it expands considerably on the material presented there. A preliminary release of our LFSC proof checker, together with the proof systems and the benchmark data discussed here are available online.[1]

1.1 Related Work

SMT proofs produced by the solver Fx7 for the AUFLIA logic of SMT-LIB have been shown to be checked efficiently by an external checker in [31]. Other approaches [19, 17] have advocated the use of interactive theorem provers to certify

[1] At `http://clc.cs.uiowa.edu/lfsc/`. The release's README file clarifies a number of minor differences with the concrete syntax used in this paper.

proofs produced by SMT solvers. The advantages of using those theorem provers are well known; in particular, their trusted core contains only a base logic and a small fixed number of proof rules. While recent works [11,2] have improved proof checking times, the performance of these tools still lags behind C/C++ checkers carefully engineered for fast checking of very large proofs. Besson *et al.* have recently proposed a similar meta-linguistic approach, though without the intention of providing a formal semantics for user-defined proof systems [9]. We are in discussion with authors of that work on how to combine LFSC with the approach they advocate.

1.2 Notational Conventions

This paper assumes some familiarity with the basics of type theory and of automated theorem proving, and will adhere in general to the notational conventions in the field. LFSC is a direct extension of LF [22], a type-theoretic logical framework based on the $\lambda\Pi$ calculus, in turn an extension of the simply typed λ-calculus. The $\lambda\Pi$ calculus has three levels of entities: values; types, understood as collections of values; and kinds, families of types. Its main feature is the support for *dependent types* which are types parametrized by values.[2] Informally speaking, if $\tau_2[x]$ is a dependent type with value parameter x, and τ_1 is a non-dependent type, the expression $\Pi x{:}\tau_1.\tau_2[x]$ denotes in the calculus the type of functions that return a value of type $\tau_2[v]$ for each value v of type τ_1 for x. When τ_2 is itself a non-dependent type, the type $\Pi x{:}\tau_1.\tau_2$ is just the arrow type $\tau_1 \to \tau_2$ of simply typed λ-calculus.

The current concrete syntax of LFSC is based on Lisp-style S-expressions, with all operators in infix format. For improved readability, we will often write LFSC expressions in abstract syntax instead. We will write concrete syntax expressions in `typewriter` font. In abstract syntax expressions, we will write variables and meta-variables in *italics* font, and constants in sans serif font. Predefined keywords in the LFSC language will be in **bold** sans serif font.

2 Introducing LF with Side Conditions

LFSC is based on the Edinburgh Logical Framework (LF) [22]. LF has been used extensively as a meta-language for encoding deductive systems including logics, semantics of programming languages, as well as many other applications [26,7, 33]. In LF, proof systems can be encoded as *signatures*, which are collections of typing declarations. Each proof rule is a constant symbol whose type represents the inferences allowed by the rule. For example, the following transitivity rule for equality

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \mathsf{eq\_trans}$$

can be encoded in LF as a constant eq_trans of type

$$\Pi t_1{:}\mathsf{term}.\, t_2{:}\mathsf{term}.\, t_3{:}\mathsf{term}.\, \Pi u_1{:}\mathsf{holds}\,(t_1 = t_2).\, \Pi u_2{:}\mathsf{holds}\,(t_2 = t_3).\, \mathsf{holds}\,(t_1 = t_3)\ .$$

[2] A simple example of dependent types is the type of bit vectors of (positive integer) size n.

The encoding can be understood intuitively as saying: for any terms t_1, t_2 and t_3, and any proofs u_1 of the equality $t_1 = t_2$ and u_2 of $t_2 = t_3$, eq_trans constructs a proof of $t_1 = t_3$. In the concrete, Lisp-style syntax of LFSC, the declaration of the rule would look like

```
(declare eq_trans (! t1 term (! t2 term (! t3 term
                  (! u1 (holds (= t1 t2)) (! u2 (holds (= t2 t3))
                   (holds (= t1 t3))))))))
```

where ! represents LF's Π binder, for the dependent function space, term and holds are previously declared type constructors, and = is a previously declared constant of type Πt_1:term. t_2:term. term (i.e., term $\rightarrow$ term $\rightarrow$ term).

Now, pure LF is not well suited for encoding large proofs from SMT solvers, due to the computational nature of many SMT inferences. For example, consider trying to prove the following simple statement in a logic of arithmetic:

$$(t_1 + (t_2 + (\ldots + t_n)\ldots)) - ((t_{i_1} + (t_{i_2} + (\ldots + t_{i_n})\ldots) = 0 \qquad (1)$$

where $t_{i_1} \ldots t_{i_n}$ is a permutation of the terms $t_1, \ldots, t_n$. A purely declarative proof would need $\Omega(n \log n)$ applications of an associativity and a commutativity rule for +, to bring opposite terms together before they can be pairwise reduced to 0.

Producing, and checking, purely declarative proofs in SMT, where input formulas alone are often measured in megabytes, is unfeasible in many cases. To address this problem, LFSC extends LF by supporting the definition of rules with side conditions, *computational checks* written in a small but expressive first-order functional language. The language has built-in types for arbitrary precision integers and rationals, inductive datatypes, ML-style pattern matching, recursion, and a very restricted set of imperative features. When checking the application of an inference rule with a side condition, an LFSC checker computes actual parameters for the side condition and executes its code. If the side condition fails, because it is not satisfied or because of an exception caused by a pattern-matching failure, the LFSC checker rejects the rule application. In LFSC, a proof of statement (1) could be given by a single application of a rule of the form:

```
(declare eq_zero (! t term (^ (normalize t) 0) (holds (= t 0))))
```

where `normalize` is the name of a separately defined function in the side condition language that takes an arithmetic term and returns a normal form for it. The expression `(^ (normalize t) 0)` defines the side condition of the eq_zero rule, with the condition succeeding if and only if the expression `(normalize t)` evaluates to `0`. We will see more about this sort of normalization rules in Section 5.2.

3 The LFSC Language and its Formal Semantics

In this section, we introduce the LFSC language in abstract syntax, by defining its formal semantics in the form of a typing relation for terms and types, and providing an operational semantics for the side-condition language. Well-typed value, type, kind and side-condition expressions are drawn from the syntactical categories defined in Figure 1 in BNF format. The kinds **type**^{c} and **type** are used to distinguish types with side conditions in them from types without.

$$
\begin{array}{llll}
\text{(Kinds)} & \kappa ::= \textsf{type} \mid \textsf{type}^{\mathrm{c}} \mid \textsf{kind} \mid \Pi x{:}\tau.\,\kappa & \text{(Types)} & \tau ::= k \mid \tau\ t \mid \Pi x{:}\tau_1[\{s\ t\}].\,\tau_2 \\
\text{(Terms)} & t\ ::= x \mid c \mid t{:}\tau \mid \lambda x[{:}\tau].\,t \mid t_1\ t_2 & \text{(Patterns)} & p\ ::= c \mid c\ x_1 \cdots x_{n+1} \\
\text{(Programs)} & s ::= x \mid c \mid -s \mid s_1 + s_2 \mid c\ s_1 \cdots s_{n+1} \mid \textsf{let}\ x\ s_1\ s_2 \mid \textsf{fail}\ \tau \mid \textsf{markvar}\ s \mid & & \\
 & \quad \textsf{ifmarked}\ s_1\ s_2\ s_3 \mid \textsf{ifneg}\ s_1\ s_2\ s_3 \mid \textsf{match}\ s\ (p_1\ s_1) \cdots (p_{n+1}\ s_{n+1}) & &
\end{array}
$$

Fig. 1 Main syntactical categories of LFSC. Letter c denotes term constants (including rational ones), x denotes term variables, k denotes type constants. The square brackets are grammar meta-symbols enclosing optional subexpressions.

Program expressions s are used in side condition code. There, we also make use of the syntactic sugar ($\textsf{do}\ s_1 \cdots s_n\ s$) for the expression ($\textsf{let}\ x_1\ s_1\ \cdots\ \textsf{let}\ x_n\ s_n\ s$) where x_1 through x_n are fresh variables. Side condition programs in LFSC are monomorphic, simply typed, first-order, recursive functions with pattern matching, inductive data types and two built-in basic types: infinite precision integers and rationals. In practice, our implementation is a little more permissive, allowing side-condition code to pattern-match also over dependently typed data. For simplicity, we restrict our attention here to the formalization for simple types only.

The operational semantics of the main constructs in the side condition language could be described informally as follows. Expressions of the form ($c\ s_1\ \cdots\ s_{n+1}$) are applications of either term constants or program constants (i.e., declared functions) to arguments. In the former case, the application constructs a new value; in the latter, it invokes a program. The expressions ($\textsf{match}\ s\ (p_1\ s_1) \cdots (p_{n+1}\ s_{n+1})$) and ($\textsf{let}\ x\ s_1\ s_2$) behave exactly as their corresponding matching and let-binding constructs in ML-like languages. The expression ($\textsf{markvar}\ s$) evaluates to the value of s if this value is a variable. In that case, the expression has also the side effect of toggling a Boolean mark on that variable.[3] The expression ($\textsf{ifmarked}\ s_1\ s_2\ s_3$) evaluates to the value of s_2 or of s_3 depending on whether s_1 evaluates to a marked or an unmarked variable. Both **markvar** and **ifmarked** raise a failure exception if their arguments do not evaluate to a variable. The expression ($\textsf{fail}\ \tau$) always raises that exception, for any type τ.

The typing rules for terms and types, given in Figure 2, are based on the rules of *canonical forms LF* [47]. They include judgments of the form $\Gamma \vdash e \Leftarrow T$ for checking that expression e has type/kind T in context Γ, where Γ, e, and T are inputs to the judgment; and judgments of the form $\Gamma \vdash e \Rightarrow T$ for computing a type/kind T for expression e in context Γ, where Γ and e are inputs and T is output. The contexts Γ map variables and constants to types or kinds, and map constants f for side condition functions to (possibly recursive) definitions of the form $(x_1 : \tau_1 \cdots x_n : \tau_n) : \tau = s$, where s is a term with free variables $x_1, \ldots, x_n$, the function's formal parameters.

The three top rules of Figure 2 define well-formed contexts. The other rules, read from conclusion to premises, induce deterministic type/kind checking and type computation algorithms. They work up to a standard definitional equality, namely $\beta\eta$-equivalence; and use standard notation for capture-avoiding substitution ($[t/x]T$ is the result of simultaneously replacing every free occurrence of x in T by t, and renaming any bound variable in T that occurs free in t). Side con-

[3] For simplicity, we limit the description here to a single mark per variable. In reality, there are 32 such marks, each with its own **markvar** command.

$$\frac{}{\cdot\ \text{Ok}} \qquad \frac{\Gamma\ \text{Ok} \quad \Gamma \vdash \tau \Rightarrow \kappa}{\Gamma,\, y : \tau\ \text{Ok}} \qquad \frac{\Gamma,\, x_1 : \tau_1, \ldots, x_n : \tau_n,\, f : k_1 \to \cdots \to k_n \to k\ \text{Ok} \quad \Gamma,\, x_1 : \tau_1, \ldots, x_n : \tau_n,\, f : k_1 \to \cdots \to k_n \to k \vdash s \Rightarrow k}{\Gamma,\, f(x_1 : k_1 \cdots x_n : k_n) : k = s\ \text{Ok}}$$

$$\frac{\Gamma\ \text{Ok}}{\Gamma \vdash \textsf{type} \Rightarrow \textsf{kind}} \qquad \frac{\Gamma\ \text{Ok}}{\Gamma \vdash \textsf{type}^{\text{c}} \Rightarrow \textsf{kind}} \qquad \frac{\Gamma\ \text{Ok} \quad y : \tau \in \Gamma}{\Gamma \vdash y \Rightarrow \tau} \qquad \frac{\Gamma \vdash t \Leftarrow \tau}{\Gamma \vdash t : \tau \Rightarrow \tau}$$

$$\frac{\Gamma \vdash \tau \Leftarrow \textsf{type} \quad \Gamma,\, x : \tau \vdash T \Rightarrow \alpha \quad \alpha \in \{\textsf{type}, \textsf{type}^{\text{c}}, \textsf{kind}\}}{\Gamma \vdash \Pi x{:}\tau.\, T \Rightarrow \alpha} \qquad \frac{\Gamma \vdash \tau \Rightarrow \Pi x{:}\tau_1.\, \kappa \quad \Gamma \vdash t \Leftarrow \tau_1}{\Gamma \vdash (\tau\ t) \Rightarrow [t/x]\kappa}$$

$$\frac{\Gamma \vdash \tau_1 \Leftarrow \textsf{type} \quad \Gamma,\, x : \tau_1 \vdash \tau_2 \Rightarrow \textsf{type} \quad \Gamma,\, x : \tau_1 \vdash s \Rightarrow \tau \quad \Gamma,\, x : \tau_1 \vdash t \Rightarrow \tau}{\Gamma \vdash \Pi x{:}\tau_1\{s\ t\}.\, \tau_2 \Rightarrow \textsf{type}^{\text{c}}}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow \Pi x{:}\tau_1.\, \tau_2 \quad \Gamma \vdash t_2 \Leftarrow \tau_1}{\Gamma \vdash (t_1\ t_2) \Rightarrow [t_2/x]\tau_2} \qquad \frac{\Gamma \vdash \tau_1 \Rightarrow \textsf{type} \quad \Gamma,\, x : \tau_1 \vdash t \Rightarrow \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, t \Rightarrow \Pi x{:}\tau_1.\, \tau_2}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow \Pi x{:}\tau_1\{s\ t\}.\, \tau_2 \quad \Gamma \vdash t_2 \Leftarrow \tau_1 \quad |\Gamma| \vdash \varepsilon;\ [t_2/y]s \downarrow [t_2/y]t;\ \sigma}{\Gamma \vdash (t_1\ t_2) \Rightarrow [t_2/x]\tau_2} \qquad \frac{\Gamma,\, x : \tau_1 \vdash t \Rightarrow \tau_2}{\Gamma \vdash \lambda x.\, t \Leftarrow \Pi x{:}\tau_1.\, \tau_2}$$

Fig. 2 Bidirectional typing rules and context rules for LFSC. Letter y denotes variables and constants declared in context Γ, letter T denotes types or kinds. Letter ε denotes the state in which every variable is unmarked.

ditions occur in type expressions of the form $\Pi y{:}\tau_1\{s\ t\}.\, \tau_2$, constructing types of kind $\textsf{type}^{\text{c}}$. The premise of the last rule, defining the well-typedness of applications involving such types, contains a judgement of the form $\Delta \vdash \sigma;\ s \downarrow s';\ \sigma'$ where Δ is a context consisting only of definitions for side condition functions, and σ and σ' are *states*, i.e., mappings from variables to their mark. Such judgment states that, under the context Δ, evaluating the expression s in state σ results in the expression s' and state σ'. In the application rule, Δ is $|\Gamma|$ defined as the collection of all the function definitions in Γ. Note that the rules of Figure 2 enforce that bound variables do not have types with side conditions in them—by requiring those types to be of kind $\textsf{type}$, as opposed to kind $\textsf{type}^{\text{c}}$. An additional requirement is not formalized in the figure. Suppose Γ declares a constant d with type $\Pi x_1{:}\tau_1.\cdots\Pi x_n{:}\tau_n.\,\tau$ of kind $\textsf{type}^{\text{c}}$, where τ is either k or $(k\ t_1\ \cdots\ t_m)$. Then neither k nor an application of k may be used as the domain of a Π-type. This is to ensure that applications requiring side condition checks never appear in types. Similar typing rules, included in the appendix, define well-typedness for side condition terms, in a fairly standard way.

A big-step operational semantics of side condition programs is provided in Figure 3 using $\Delta \vdash \sigma;\ s \downarrow s';\ \sigma'$ judgements. For brevity, we elide "$\Delta \vdash$" from the rules when Δ is unused. Note that side condition programs can contain unbound variables, which evaluate to themselves. States σ (updated using the notation $\sigma[x \mapsto v]$) map such variables to the value of their Boolean mark. If no rule applies when running a side condition, program evaluation and hence checking of types with side conditions fails. This also happens when evaluating the fail construct ($\textsf{fail}\ \tau$), or when pattern matching fails. Currently, we do not enforce termination of side condition programs, nor do we attempt to provide facilities to reason formally about the behavior of such programs.

Our implementation of LFSC supports the use of the wildcard symbol _ in place of an actual argument of an application when the value of this argument is

$$\frac{}{\sigma_1;\, c \downarrow c;\, \sigma_1} \qquad \frac{}{\sigma_1;\, x \downarrow x;\, \sigma_1} \qquad \frac{\sigma_1;\, s \downarrow x;\, \sigma_2}{\sigma_1;\, (\textbf{markvar}\ s) \downarrow x;\, \sigma_2[x \mapsto \neg\sigma_2(x)]}$$

$$\frac{\sigma_1;\, s_1 \downarrow r;\, \sigma_2 \quad r < 0 \quad \sigma_2;\, s_2 \downarrow s_2';\, \sigma_3}{\sigma_1;\, (\textbf{ifneg}\ s_1\ s_2\ s_3) \downarrow s_2';\, \sigma_3} \qquad \frac{\sigma_1;\, s_1 \downarrow r;\, \sigma_2 \quad r \geq 0 \quad \sigma_2;\, s_3 \downarrow s_3';\, \sigma_3}{\sigma_1;\, (\textbf{ifneg}\ s_1\ s_2\ s_3) \downarrow s_3';\, \sigma_3}$$

$$\frac{\sigma_1;\, s_1 \downarrow x;\, \sigma_2 \quad \sigma_2(x) \quad \sigma_2;\, s_2 \downarrow s_2';\, \sigma_3}{\sigma_1;\, (\textbf{ifmarked}\ s_1\ s_2\ s_3) \downarrow s_2';\, \sigma_3} \qquad \frac{\sigma_1;\, s_1 \downarrow x;\, \sigma_2 \quad \neg\sigma_2(x) \quad \sigma_2;\, s_3 \downarrow s_3';\, \sigma_3}{\sigma_1;\, (\textbf{ifmarked}\ s_1\ s_2\ s_3) \downarrow s_3';\, \sigma_3}$$

$$\frac{\sigma_1;\, s_1 \downarrow s_1';\, \sigma_2 \quad \sigma_2;\, [s_1'/x]s_2 \downarrow s_2';\, \sigma_3}{\sigma_1;\, (\textbf{let}\ x\ s_1\ s_2) \downarrow s_2';\, \sigma_3} \qquad \frac{\forall i \in \{1, \ldots, n\}, (\sigma_i;\, s_i \downarrow s_i';\, \sigma_{i+1})}{\sigma_1;\, (c\ s_1\ \cdots\ s_n) \downarrow (c\ s_1'\ \cdots\ s_n');\, \sigma_{n+1}}$$

$$\frac{\sigma_1;\, s \downarrow (c\ s_1'\ \cdots\ s_n');\, \sigma_2 \quad \exists i\, p_i = (c\ x_1\ \cdots\ x_n) \quad \sigma_2;\, [s_1'/x_1, \ldots, s_n'/x_n]s_i \downarrow s';\, \sigma_3}{\sigma_1;\, (\textbf{match}\ s\ (p_1\ s_1)\ \cdots\ (p_m\ s_m)) \downarrow s';\, \sigma_3}$$

$$\frac{\begin{array}{l}\forall i \in \{1, \ldots, n\}\, (\Delta \vdash \sigma_i;\ s_i \downarrow s_i';\, \sigma_{i+1}) \\ (f(x_1 : \tau_1\ \cdots\ x_n : \tau_n) : \tau = s) \in \Delta \qquad \Delta \vdash \sigma_{n+1};\ [s_1'/x_1, \ldots, s_n'/x_n]s \downarrow s';\, \sigma_{n+2}\end{array}}{\Delta \vdash \sigma_1;\ (f\ s_1\ \cdots\ s_n) \downarrow s';\, \sigma_{n+2}}$$

Fig. 3 Operational semantics of side condition programs. We omit the straightforward rules for the rational operators $-$ and $+$.

determined by the types of later arguments. This feature, which is analogous to implicit arguments in theorem provers such as Coq and programming languages such as Scala, is crucial to avoid bloating proofs with redundant information. In a similar vein, the concrete syntax allows a form of lambda abstraction that does not annotate the bound variable with its type when that type can be computed efficiently from context.

We conclude by pointing out that LFSC's type system is a refinement of LF's, in the following sense. Let $||\tau||$ denote the type obtained from τ by erasing any side condition constraints from the Π-abstractions in τ; let $||\textbf{type}^c||$ be **type**; and extend this notation to contexts in the natural way. Then, we have the following.

Theorem 1 *For all Γ, t and τ, if $\Gamma \vdash t{:}\tau$ in LFSC, then $||\Gamma|| \vdash t{:}||\tau||$ in LF.*

Proof. By a straightforward induction on LFSC typing derivations. ■

4 Encoding propositional and core SMT reasoning in LFSC

In this section and the next, we illustrate the power and flexibility of LFSC for SMT proof checking by discussing a number of proof systems relevant to SMT, and their possible encodings in LFSC. Our goal is not to be exhaustive, but to provide representative examples of how LFSC allows one to encode a variety of logics and proof rules while paying attention to proof checking performance issues. Section 6 focuses on the latter by reporting on our initial experimental results.

Roughly speaking, proofs generated by SMT solvers, especially those based on the DPLL(T) architecture [36], are two-tiered refutation proofs, with a propositional skeleton filled with several theory-specific subproofs [20]. The conclusion, a trivially unsatisfiable formula, is reached by means of propositional inferences applied to a set of input formulas and a set of *theory lemmas*. These are disjunctions

```
(declare var type)                  (declare clause type)
(declare lit type)                  (declare cln clause)
(declare pos (! x var lit))         (declare clc (! l lit (! c clause clause)))
(declare neg (! x var lit))
```

Fig. 4 Definition of propositional clauses in LFSC concrete syntax.

of theory literals proved from no assumptions mostly with proof rules specific to the theory or theories in question—the theory of real arithmetic, of arrays, etc.

Large portions of the proof's propositional part consist typically of applications of some variant of the resolution rule. These subproofs are generated similarly to what is done by proof-producing SAT solvers, where resolution is used for conflict analysis and lemma generation [51, 20]. A proof format proposed in 2005 by Van Gelder for SAT solvers is based directly on resolution [46]. Input formulas in SMT differ from those given to SAT solvers both for being not necessarily in Conjunctive Normal Form and for having non-propositional atoms. As a consequence, the rest of the propositional part of SMT proofs involve CNF conversion rules as well as *abstraction* rules that uniformly replace theory atoms in input formulas and theory lemmas with Boolean variables. While SMT solvers usually work just with quantifier-free formulas, some of them can reason about quantifiers as well, by generating and using selected ground instances of quantified formulas. In these cases, output proofs also contain applications of rules for quantifier instantiation.

In the following, we demonstrate different ways of representing propositional clauses and SMT formulas and lemmas in LFSC, and of encoding proof systems for them with various degrees of support for efficient proof checking. For simplicity and space constraints, we consider only a couple of individual theories, and restrict our attention to quantifier-free formulas. We note that encoding proofs involving combinations of theories is more laborious but not qualitatively more difficult; encoding SMT proofs for quantified formulas is straightforward thanks to LFSC's support for higher-order abstract syntax which allows one to represent and manipulate quantifiers as higher-order functions, in a completely standard way.[4]

4.1 Encoding propositional resolution

The first step in encoding any proof system in LFSC (or LF for that matter) is to encode its formulas. In the case of propositional resolution, this means encoding propositional clauses. Figure 4 presents a possible encoding, with type and type constructor declarations in LFSC's concrete syntax. We first declare an LFSC type var for propositional variables and then a type lit for propositional literals. Type lit has two constructors, pos and neg, both of type $\Pi x{:}\mathsf{var}.\,\mathsf{lit}$[5] which turn a variable into a literal of positive, respectively negative, polarity. We use these to represent positive and negative occurrences of a variable in a clause. The type clause, for

[4] For instance $\forall x{:}\tau.\,\phi$ can be represented as $(\mathsf{forall}\ \lambda x{:}\tau.\,\phi)$ where forall is a constant of type $(\tau \to \mathsf{formula}) \to \mathsf{formula}$. Then, quantifier instantiation reduces to (lambda-term) application.

[5] Recall that the ! symbol in the concrete syntax stands for Π-abstraction.

```
(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))
          (drop_dups d))))))

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

Fig. 5 The propositional resolution calculus in LFSC concrete syntax.

propositional clauses, is endowed with two constructors that allow the encoding of clauses as lists of literals. The constant cln represents the empty clause ($\Box$). The function clc intuitively takes a literal l and a clause c, and returns a new clause consisting of l followed by the literals of c. For an example, a clause like $P \vee \neg Q$ can be encoded as the term (clc (pos P) (clc (neg Q) cln)).

Figure 5 provides LFSC declarations that model binary propositional resolution with factoring. The type holds, indexed by values of type clause, represents the type of proofs for clauses. Intuitively, for any clause c, values of type (holds c) are proofs of c. The side-condition function resolve takes two clauses and a variable v, and returns the result of resolving the two clauses together with v as the pivot[6], after eliminating any duplicate literals in the resolvent. The constructor R encodes the resolution inference rule. Its type

$$\Pi c_1\text{:clause}.\ \Pi c_2\text{:clause}.\ \Pi c_3\text{:clause}.\\ \Pi u_1\text{:holds } c_1.\ \Pi u_2\text{:holds } c_2.\ \Pi v\text{:var } \{(\text{resolve } c_1\ c_2\ v)\ c_3\}.\ \text{holds } c_3$$

can be paraphrased as follows: for any clauses c_1, c_2, c_3 and variables v, the rule R returns a proof of c_3 from a proof of c_1 and a proof of c_2 *provided that* c_3 is the result of successfully applying the resolve function to c_1, c_2 and v. The side condition function resolve is defined as follows (using a number of auxiliary functions whose definition can be found in the appendix). To resolve clauses c_1 and c_2 with pivot v, v must occur in a positive literal of c_1 and a negative literal of c_2 (checked with the in function). In that case, the resolvent clause is computed by removing (with remove) all positive occurrences of v from c_1 and all negative ones from c_2, concatenating the resulting clauses (with append), and finally dropping any duplicates from the concatenation (with drop_dups); otherwise, resolve, and consequently the side condition of R, fails.

In proof terms containing applications of the R rule, the values of its input variables c_1, c_2 and c_3 can be determined from later input values, namely the concrete types of u_1, u_2 and v, respectively. Hence, in those applications $c_1, \ldots, c_3$ can be replaced by the wildcard _, as mentioned in Section 3 and shown in Figure 6.

The single rule above is enough to encode proofs in the propositional resolution calculus. This does not appear to be possible in LF. Without side conditions one also needs auxiliary rules, for instance, to move a pivot to the head of the list representing the clause and to perform factoring on the resolvent. The upshot of this

[6] A variable v is the *pivot* of a resolution application with resolvent $c_1 \vee c_2$ if the clauses resolved upon are $c_1 \vee v$ and $\neg v \vee c_2$.

$$\dfrac{\dfrac{V_1 \vee V_2 \quad \neg V_1 \vee V_2}{V_2} \quad \dfrac{\neg V_2 \vee V_3 \quad \neg V_3 \vee \neg V_2}{\neg V_2}}{\square}$$

λv_1:var. λv_2:var. λv_2:var.
λp_1:holds $(v_1 \vee v_2)$. λp_2:holds $(\neg v_1 \vee v_2)$.
λp_3:holds $(\neg v_2 \vee v_3)$. λp_4:holds $(\neg v_3 \vee \neg v_2)$.
(R _ _ _ (R _ _ _ p_1 p_2 v_1) (R _ _ _ p_3 p_4 v_3) v_2) : holds $\square$)

```
(check
 (% v1 var (% v2 var (% v3 var
  (% p1 (holds (clc (pos v1) (clc (pos v2) cln)))
   (% p2 (holds (clc (neg v1) (clc (pos v2) cln)))
    (% p3 (holds (clc (neg v2) (clc (pos v3) cln)))
     (% p4 (holds (clc (neg v3) (clc (neg v2) cln)))
      (: (holds cln) (R _ _ _ (R _ _ _ p1 p2 v1) (R _ _ _ p3 p4 v3) v2)))))))))))
```

Fig. 6 An example refutation and its LFSC encoding, respectively in abstract and in concrete syntax (as argument of the `check` command). In the concrete syntax, `(% x τ t)` stands for $\lambda x{:}\tau.\, t$; for convenience, the ascription operator `:` takes first a type and then a term.

is a more complex proof system and bigger proofs. Other approaches to checking resolution proofs avoid the need for those auxiliary rules by hard coding the clause type in the proof checker and implementing it as a set of literals. An example is work by Weber and Amjad on reconstructing proofs produced by an external SAT solver in Isabelle/HOL [48]. They use several novel encoding techniques to take advantage of the fact that the native sequents of the Isabelle/HOL theorem prover are of the form $\Gamma \vdash \phi$, where Γ is interpreted as a set of literals. They note the importance of these techniques for achieving acceptable performance over their earlier work, where rules for reordering literals in a clause, for example, were required. Their focus is on importing external proofs into Isabelle/HOL, not trustworthy efficient proof-checking in its own right. But we point out that it would be wrong to conclude that their approach is intrinsically more declarative than the LFSC approach: in their case, the computational side-conditions needed to maintain the context Γ as a set have simply been made implicit, as part of the core inference system of the theorem prover. In contrast, the LFSC approach makes such side conditions explicit, and user-definable.

Example 1 For a simple example of a resolution proof, consider a propositional clause set containing the clauses $c_1 := \neg V_1 \vee V_2$, $c_2 := \neg V_2 \vee V_3$, $c_3 := \neg V_3 \vee \neg V_2$, and $c_4 := V_1 \vee V_2$. A resolution derivation of the empty clause from these clauses is given in Figure 6. The proof can be represented in LFSC as the lambda term below the proof tree. Ascription is used to assign type (holds $\square$) to the main subterm (R _ ... v_2) under the assumption that all four input clauses hold. This assumption is encoded by using the input (i.e., lambda) variables $p_1, \ldots, p_4$ of type (holds c_1), ..., (holds c_4), respectively. Checking the correctness of the original proof in the resolution calculus then amounts to checking that the lambda term is well-typed in LFSC when its _ holes are filled in as prescribed by the definition of R. In the concrete syntax, this is achieved by passing the proof term to the check command. ■

The use of lambda abstraction in the example above comes from standard LF encoding methodology. In particular, note how object-language variables (the V_i's)

```
(declare clr (! l lit (! c clause clause)))
(declare con (! c1 clause (! c2 clause clause)))

(declare DR (! c1 clause (! c2 clause (! u1 (holds c1) (! u2 (holds c2) (!v var
             (holds (con (clr (pos v) c1) (clr (neg v) c2)))))))))

(declare S (! c1 clause (! c2 clause (! u (holds c1) (^ (simplify c1) c2)
            (holds c2)))))
```

Fig. 7 New clause type and rules for deferred resolution.

fun simplify (x : clause) : clause =
  match x **with**
    cln $\rightarrow$ cln
    con c_1 c_2 $\rightarrow$ append (simplify c_1) (simplify c_2)
    clc l c $\rightarrow$
      if l is marked for deletion **then** (simplify c)
      else mark l for deletion; d = clc l (simplify c); unmark l; d
    clr l c $\rightarrow$
      if l is marked for deletion **then** d = simplify c
      else mark l for deletion; d = simplify c; unmark l;
      if l was deleted from c **then** d **else fail**

Fig. 8 Pseudo-code for side condition function used by the S rule.

are represented by LFSC meta-variables (the λ-variables $v_1, \ldots, v_4$). This way, safe renaming and safe substitution of bound variables at the object level are inherited for free from the meta-level. In LFSC, an additional motivation for using meta-variables for object language variables is that we can efficiently test the former for equality in side conditions using variable marking. In the resolution proof system described here, this is necessary in the side condition of the R rule—for instance, to check that the pivot occurs in the clauses being resolved upon (see Appendix B).

4.2 Deferred Resolution

The single rule resolution calculus presented above can be further improved in terms of proof checking performance by delaying the side condition tests, as done in constrained resolution approaches [41]. One can modify the clause data structure so that it includes constraints representing those conditions. Side condition constraints are accumulated in resolvent clauses and then checked periodically, possibly just at the very end, once the final clause has been deduced. The effect of this approach is that (i) checking resolution applications becomes a constant time operation, and (ii) side condition checks can be deferred, accumulated, and then performed more efficiently in a single sweep using a new rule that converts a constrained clause to a regular one after discharging its attached constraint.

There are many ways to implement this general idea. We present one in Figure 7, based on extending the clause type of Figure 4 with two more constructors: clr and con. The term (clr l c) denotes the clause consisting of all the literals of c except l, assuming that l indeed occurs in c. The expression (con c_1 c_2) denotes the clause consisting of all the literals that are in c_1 or in c_2. Given two clauses c_1 and c_2 and a pivot variable v, the new resolution rule DR, with no side condi-

tions, produces the resolvent (con (clr (pos v) c_1) (clr (neg v) c_2)) which carries within itself the *resolution constraint* that (pos v) must occur in c_1 and (neg v) in c_2. Applications of the resolution rule can alternate with applications of the rule S, which converts a resolvent clause into a regular clause (constructed with just cln and clc) while also checking that the resolvent's resolution constraints are satisfied. A sensible strategy is to apply S both to the final resolvent and to any intermediate resolvent that is used more than once in the overall proof—to avoid unnecessary duplication of constraints.

The side condition function for S is provided in pseudo-code (for improved readability) in Figure 8. The pseudo-code should be self-explanatory. The auxiliary function append, defined only on regular clauses, works like a standard list append function. Since the cost of append is linear in the first argument, simplify executes more efficiently with linear resolution proofs, where at most one of the two premises of each resolution step is a previously proved (and simplified) lemma. Such proofs are naturally generated by SMT solvers with a propositional engine based on conflict analysis and lemma learning—which means essentially all SMT solvers available today. In some cases, clauses returned by simplify may contain duplicate literals. However, such literals will be removed by subsequent calls to simplify, thereby preventing any significant accumulation in the clauses we produce.

Our experiments show that deferred resolution leads to significant performance improvements at proof checking time: checking deferred resolution proofs is on average 5 times faster than checking proofs using the resolution rule R [38]. The increased speed does come here at the cost of increased size and complexity of the side condition code, and so of the trusted base. The main point is again that LFSC gives users the choice of how big they want the trusted base to be, while also documenting that choice explicitly in the side condition code.

4.3 CNF Conversion

Most SMT solvers accept as input quantifier-free formulas (from now on simply *formulas*) but do the bulk of their reasoning on a set of clauses derived from the input via a conversion to CNF or, equivalently, clause form. For proof checking purposes, it is then necessary to define proof rules that account for this conversion. Defining a good set of such proof rules is challenging because of the variety of CNF transformations used in practice. Additional difficulties, at least when using logical frameworks, come from more mundane but nevertheless important problems such as how to encode with proof rules, which have a fixed number of premises, transformations that treat operators like logical conjunction and disjunction as multiarity symbols, with an arbitrary number of arguments.

To show how these difficulties can be addressed in LFSC we discuss now a hybrid data structure we call *partial clauses* that mixes formulas and clauses and supports the encoding of many CNF conversion methods as small step transformations on partial clauses. Partial clauses represent intermediate states between an initial formula to be converted to clause form and its final clause form. We then present a general set of rewrite rules on partial clauses that can be easily encoded as LFSC proof rules. Abstractly, a partial clause is simply a pair

$$(\phi_1, \ldots, \phi_m;\ l_1 \vee \cdots \vee l_n)$$

$$
\begin{array}{lrcll}
\mathsf{dist\_pos} & (\phi_1 \wedge \phi_2, \Phi;\ c) & \Longrightarrow & (\phi_1, \Phi;\ c), (\phi_2, \Phi;\ c) & \\
\mathsf{dist\_neg} & (\neg(\phi_1 \wedge \phi_2), \Phi;\ c) & \Longrightarrow & (\neg\phi_1, \neg\phi_2, \Phi;\ c) & \\
\mathsf{flat\_pos} & (\phi_1 \vee \phi_2, \Phi;\ c) & \Longrightarrow & (\phi_1, \phi_2, \Phi;\ c) & \\
\mathsf{flat\_neg} & (\neg(\phi_1 \vee \phi_2), \Phi;\ c) & \Longrightarrow & (\neg\phi_1, \Phi;\ c),\ (\neg\phi_2, \Phi;\ c) & \\
\mathsf{rename} & (\phi, \Phi;\ c) & \Longrightarrow & (\Phi;\ v, c),\ (\phi;\ \neg v),\ (\neg\phi;\ v) & (v \text{ is a fresh var})
\end{array}
$$

Fig. 9 Sample CNF conversion rules for partial clauses (shown as a rewrite system). Φ is a sequence of formulas and c is a sequence of literals (a clause).

```
(declare formula type)      (declare not (! phi formula formula)
(declare formSeq type)      (declare empty formSeq)
                            (declare ins (! phi formula (! Phi formSeq formSeq)))

(declare pc_holds (! Phi formSeq (! c clause type)))

(declare rename (! phi formula (! Phi formSeq (! c clause
 (! q (pc_holds (ins phi Phi) c)
 (! r (! v var (! r1 (pc_holds Phi (clc (pos v) c))
               (! r2 (pc_holds (ins phi empty) (clc (neg v) cln))
               (! r3 (pc_holds (ins (not phi) empty) (clc (pos v) cln))
                (holds cln)))))
  (holds cln)))))))
```

Fig. 10 LFSC proof rule for rename transformation in Figure 9.

consisting of a (possibly empty) sequence of formulas and a clause. Semantically, it is just the disjunction $\phi_1 \vee \cdots \vee \phi_m \vee l_1 \vee \cdots \vee l_n$ of all the formulas in the sequence with the clause. A set $\{\phi_1, \ldots, \phi_k\}$ of input formulas, understood conjunctively, can be represented as the sequence of partial clauses $(\phi_1;\ ), \ldots, (\phi_k;\ )$. A set of rewrite rules can be used to turn this sequence into an equisatisfiable sequence of partial clauses of the form $(\ ;\ c_1), \ldots, (\ ;\ c_n)$, which is in turn equisatisfiable with $c_1 \wedge \cdots \wedge c_n$. Figure 9 describes some of the rewrite rules for partial clauses. We defined 31 CNF conversion rules to transform partial clauses. Most rules eliminate logical connectives and let-bindings in a similar way as the ones shown in Figure 9. Several kinds of popular CNF conversion algorithms can be realized as particular application strategies for this set of rewrite rules (or a subset thereof).

Formulating the rewrite rules of Figure 9 into LFSC proof rules is not difficult. The only challenge is that conversions based on them and including rename are only satisfiability preserving, not equivalence preserving. To guarantee soundness in those cases we use natural-deduction style proof rules of the following general form for each rewrite rule $p \Longrightarrow p_1, \ldots, p_n$ in Figure 9: derive $\Box$, the empty clause, from (i) a proof of the partial clause p and (ii) a proof of $\Box$ from the partial clauses $p_1, \ldots, p_n$. We provide one example of these proof rules in Figure 10, namely the one for rename; the other proof rules are similar. In the figure, the type formSeq for sequences of formulas has two constructors, analogous to the usual ones for lists. The constructor pc_holds is the analogous of holds, but for partial clauses—it denotes a proof of the partial clause $(\Phi;\ c)$ for every sequence Φ of formulas and clause c. Note how the requirement in rename that the variable v be fresh is achieved at the meta-level in the LFSC proof rule with the use of a Π-bound variable.

```
(declare th_holds (! phi formula type))

(declare assume_true                    (declare assume_false
 (! v var (! phi formula (! c clause     (! v var (! phi formula (! c clause
  (! r (atom v phi)                       (! r (atom v phi)
   (! u (! o (th_holds phi) (holds c))     (! u (! o (th_holds (not phi)) (holds c))
    (holds (clc (neg v) c))))))))           (holds (clc (pos v) c))))))))
```

Fig. 11 Assumption rules for theory lemmas in LFSC concrete syntax

4.4 Converting Theory Lemmas to Propositional Clauses

When converting input formulas to clause form, SMT solvers also abstract each *theory atom* ϕ (e.g., $s = t, s < t$, etc.) occurring in the input with a unique propositional variable v, and store the corresponding mapping internally. This operation can be encoded in LFSC using a proof rule similar to rename from Figure 10, but also incorporating the mapping between v and ϕ. In particular, SMT solvers based on the *lazy approach* [42,4] abstract theory atoms with propositional variables to separate propositional reasoning, done by a SAT engine which works with a set of propositional clauses, from theory reasoning proper, done by an internal *theory solver* which works only with sets of *theory literals*, theory atoms and their negations. At the proof level, the communication between the theory solver and the SAT engine is established by having the theory solver prove some theory lemmas, in the form of disjunctions of theory literals, whose abstraction is then used by the SAT engine as if it was an additional input clause. A convenient way to produce proofs that connect proofs of theory lemmas with Boolean refutations, which use abstractions of theory lemmas and of clauses derived from the input formulas, is again to use natural deduction-style proof rules.

Figure 11 shows two rules used for this purpose. The rule assume_true derives the propositional clause $\neg v \vee c$ from the assumptions that (i) v abstracts a formula ϕ (expressed by the type (atom v ϕ)) and (ii) c is provable from ϕ. Similarly, assume_false derives the clause $v \vee c$ from the assumptions that v abstracts a formula ϕ and c is provable from $\neg\phi$. Suppose $\psi_1 \vee \cdots \vee \psi_n$ is a theory lemma. A proof-producing theory solver can be easily instrumented to prove the empty clause from the assumptions $\overline{\psi}_1, \ldots, \overline{\psi}_n$, where $\overline{\psi}_i$ denotes the complement of the literal ψ_i. This proof can be expressed by the theory solver with nested applications of assume_true and assume_false, and become a proof of the propositional clause $l_1 \vee \cdots \vee l_n$, where each l_i is the propositional literal corresponding to ψ_i.

Example 2 Consider a theory lemma such as $\neg(s = t) \vee t = s$, say, for some terms s and t. Staying at the abstract syntax level, let P be a proof term encoding a proof of $\Box$ from the assumptions $s = t$ and $\neg(t = s)$. By construction, this proof term has type (holds $\Box$). Suppose a_1, a_2 are meta-variables of type (atom v_1 $(s = t)$) and (atom v_2 $(t = s)$), respectively, for some meta-variables v_1 and v_2 of type var. Then, the proof term

$$\begin{array}{l}(\mathsf{assume\_true}\ \_\ \_\ \_\ a_1\ (\lambda h_1.\\ \ (\mathsf{assume\_false}\ \_\ \_\ \_\ a_2\ (\lambda h_2.\, P)))\end{array}$$

$$\frac{x - x \leq c \quad \{c < 0\}}{\square}\ \textsf{idl\_contra} \qquad \frac{x - y < c \quad \{c - 1 = d\}}{x - y \leq d}\ \textsf{lt\_to\_leq}$$

$$\frac{x - y \leq a \quad y - z \leq b \quad \{a + b = c\}}{x - z \leq c}\ \textsf{idl\_trans}$$

Fig. 12 Sample QF_IDL rules and LFSC encodings (x, y, z are constant symbols and a, b, c, d are integer values.)

```
(declare int type)                    (declare as_int (! x mpz int))

(declare idl_contra (! x int (! c mpz
 (! u (th_holds (<= (- x x) (as_int c))) (^ (ifneg c tt ff) tt)
  (holds cln)))))
```

Fig. 13 LFSC encoding of the idl_contra rule. Type mpz is the built-in arbitrary precision integer type (the name comes from the underlying GNU Multiple Precision Arithmetic Library, `libgmp`); as_int builds a term of the SMT integer type int from a mpz number; tt and ff are the constructors of the `bool` predefined type for Booleans; (ifneg x y z) evaluates to y or z depending on whether the mpz number x is negative or not.

has type holds ($\neg v_1 \vee v_2$) and can be included in larger proof terms declaring v_1, v_2, a_1, and a_2 as above. Note that the λ-variables h_1 and h_2 do not need a type annotation here as their types can be inferred from the types of a_1 and a_2. ■

5 Encoding SMT logics

In this section, we show in more detail how LFSC allows one to represent SMT proofs involving theory reasoning. We consider two basic logics (i.e., fragments of first-order theories) in the SMT-LIB standard [5]: QF_IDL and QF_LRA.

5.1 Quantifier-Free Integer Difference Logic

The logic QF_IDL, for *quantifier-free integer difference logic*, consists of formulas interpreted over the integer numbers and restricted (in essence) to Boolean combinations of atoms of the form $x - y \leq c$ where x and y are integer variables (equivalently, free constants) and c is an integer value, i.e., a possibly negated numeral. Some QF_IDL rules for reasoning about the satisfiability of sets of literals in this logic are shown in Figure 12, in conventional mathematical notation. Rule side conditions are provided in braces, and are to be read as semantic expressions; for example, $a + b = c$ in a side condition should be read as "c is the result of adding a and b." Note that the side conditions involve only values, and so can be checked by (simple) computation. The actual language QF_IDL contains additional atoms besides those of the form of $x - y \leq c$. For instance, atoms such as $x < y$ and $x - y \geq c$ are also allowed. Typical solvers for this logic use then a number of normalization rules to reduce these additional atoms to the basic form. An example would be rule lt_to_leq in Figure 12.

Encoding typical QF_IDL proof rules in LFSC is straightforward thanks to the built-in support for side conditions and for arbitrary precision integers in the side condition language. As an example, Figure 13 shows the idl_contra rule.

$$\frac{p_1 = 0 \quad p_2 \sim 0}{(p_1 + p_2)\downarrow \sim 0}\ \mathsf{lra\_add}\ {=}{\sim} \qquad \frac{p_1 \sim 0 \quad p_2 = 0}{(p_1 - p_2)\downarrow \sim 0}\ \mathsf{lra\_sub}\ {\sim}{=} \qquad \frac{\{a \sim 0\}}{a \sim 0}\ \mathsf{lra\_axiom}\ \sim$$

$$\frac{p = 0}{(a \cdot p)\downarrow = 0}\ \mathsf{lra\_mult\_c}\ = \qquad \frac{p > 0 \quad \{a > 0\}}{(a \cdot p)\downarrow > 0}\ \mathsf{lra\_mult\_c}\ > \qquad \frac{p \sim 0 \quad \{p \nsim 0\}}{\square}\ \mathsf{lra\_contra}\ \sim$$

Fig. 14 Some of proof rules for linear polynomial atoms. Letter p denotes normalized linear polynomials; a denotes rational values; the arithmetic operators denote operations on linear polynomials.

5.2 Quantifier-Free Linear Real Arithmetic

The logic QF_LRA, for *quantifier-free linear arithmetic*, consists of formulas interpreted over the real numbers and restricted to Boolean combinations of linear equations and inequations over real variables, with rational coefficients. We sketch an approach for encoding, in LFSC, refutations for sets of QF_LRA literals. We devised an LFSC proof system for QF_LRA that centers around proof inferences for *normalized linear polynomial atoms*; that is, atoms of the form

$$a_1 \cdot x_1 + \ldots + a_n \cdot x_n + a_{n+1} \sim 0$$

where each a_i is a rational value, each x_i is a real variable, and $\sim$ is one of the operators $=, >, \geq$. We represent linear polynomials in LFSC as (inductive data type) values of the form (pol a l), where a is a rational value and l is a list of monomials of the form (mon a_i x_i) for rational value a_i and real variable x_i. With this representation, certain computational inferences in QF_LRA become immediate. For example, to verify that a normalized linear polynomial (pol a l) is the zero polynomial, it suffices to check that a is zero and l is the empty list. With normalized linear polynomials, proving a statement like (1) in Section 2 amounts to a single rule application whose side condition is an arithmetic check for equality between rational values.

Figure 14 shows a representative selection of the rules in our LFSC proof system for QF_LRA based on a normalized linear polynomial representation of QF_LRA literals.[7] Again, side conditions are written together with the premises, in braces, and are to be read as mathematical notation. For example, the side condition $\{p + p' = 0\}$, say, denotes the result of checking whether the expression $p + p'$ evaluates to the zero element of the polynomial ring $\mathbb{Q}[X]$, where $\mathbb{Q}$ is the field of rational numbers and X the set of all variables. Expression of the form $e\downarrow$ in the rules denote the result of normalizing the linear polynomial expression e. The normalization is actually done in the rule's side condition, which is however left implicit here to keep the notation uncluttered. The main idea of this proof system, based on well known results, is that a set of normalized linear polynomial atoms can be shown to be unsatisfiable if and only if it is possible to multiply each of them by some rational coefficient such that their sum normalizes to an atom of the form $p \sim 0$ that does not hold in $\mathbb{Q}[X]$.

Since the language of QF_LRA is not based on normalized linear polynomial atoms, any refutation for a set of QF_LRA literals must also include rules that account for the translation of this set into an equisatisfiable set of normalized

[7] Additional cases are omitted for the sake of brevity. A comprehensive list of rules can be found in the appendix of [39].

$$\frac{t_1 = p_1 \quad t_2 = p_2}{t_1 + t_2 = (p_1 + p_2)\downarrow}\ \mathsf{pol\_norm}_+ \qquad \frac{t = p}{a_\mathrm{t} \cdot t = (a_\mathrm{p} \cdot p)\downarrow}\ \mathsf{pol\_norm}_{\mathsf{c}\cdot} \qquad \frac{}{a_\mathrm{t} = a_\mathrm{p}}\ \mathsf{pol\_norm\_const}$$

$$\frac{t_1 = p_1 \quad t_2 = p_2}{t_1 - t_2 = (p_1 - p_2)\downarrow}\ \mathsf{pol\_norm}_- \qquad \frac{t = p}{t \cdot a_\mathrm{t} = (p \cdot a_\mathrm{p})\downarrow}\ \mathsf{pol\_norm}_{\cdot\mathsf{c}} \qquad \frac{}{v_\mathrm{t} = v_\mathrm{p}}\ \mathsf{pol\_norm\_var}$$

$$\frac{t_1 \sim t_2 \quad t_1 - t_2 = p}{p \sim 0}\ \mathsf{pol\_norm}_\sim$$

Fig. 15 Proof rules for conversion to normalized linear polynomial atoms. Letter t denotes QF_LRA terms; a_t and a_p denote the same rational constant, in one case considered as a term and in the other as a polynomial (similarly for the variables v_t and v_p).

```
(declare term type)
(declare poly type)   (declare pol_norm (! t term (! p poly type)))

(declare pol_norm_+ (! t1 term (! t2 term (! p1 poly (! p2 poly (! p3 poly
 (! pn1 (pol_norm t1 p1) (! pn2 (pol_norm t2 p2) (^ (poly_add p1 p2) p3)
  (pol_norm (+ t1 t2) p3)))))))))
```

Fig. 16 Normalization function for + terms in concrete syntax.

linear polynomial atoms. Figure 15 provides a set of proof rules for this translation. Figure 16 shows, as an example, the encoding of rule $\mathsf{pol\_norm}_+$ in LFSC's concrete syntax. The type (pol_norm t p) encodes the judgment that p is the polynomial normalization of term t—expressed in the rules of Figure 15 as $t = p$. In $\mathsf{pol\_norm}_+$, the polynomials p_1 and p_2 for the terms t_1 and t_2 are added together using the side condition poly_add, which produces the normalized polynomial $(p_1 + p_2)\downarrow$. The side condition of the rule requires that polynomial to be the same as the provided one, p_3. The other proof rules are encoded in a similar way using side condition functions that implement the other polynomial operations (subtraction, scalar multiplication, and comparisons between constant polynomials).

We observe that the overall size of the side condition code in this proof system is rather small: about 60 lines total (and less than 2 kilobytes). Complexity-wise, the various normalization functions are comparable to a merge sort of lists of key/value pairs. As a result, manually verifying the correctness of the side conditions is fairly easy.

6 Empirical Results

In this section we provide some empirical results on LFSC proof checking for the logics presented in the previous section. These results were obtained with an LFSC proof checker that we have developed to be both general (i.e., supporting the whole LFSC language) and fast. The tool, which we will refer to as LFSC here, is more accurately described as a *proof checker generator*: given an LFSC signature, a text file declaring a proof system in LFSC format, it produces a checker for that proof system. Some of its notable features in support of high performance proof checking are the compilation of side conditions, as opposed to the incorporation of a side condition language interpreter in the proof checker, and the generation of proof checkers that check proof terms on the fly, as they parse them. See previous

work by Stump et al, as well as work by Necula et al., for more on fast LF proof checking [43, 50, 44, 35, 34].

6.1 QF_IDL

In separate work, we developed an SMT solver for the QF_IDL logic, mostly to experiment with proof generation in SMT. This solver, called CLSAT, can solve moderately challenging QF_IDL benchmarks from the SMT-LIB library, namely those classified with difficulty 0 through 3.[8] We ran CLSAT on the unsatisfiable QF_IDL benchmarks in SMT-LIB, and had it produce proofs in the LFSC proof system sketched in Section 5.1 optimized with the deferred resolution rule described in Section 4.2. Then we checked the proofs using the LFSC checker. The experiments were performed on the SMT-EXEC solver execution service.[9] A timeout of 1,800 seconds was used for each of the 622 benchmarks. The table below summarizes those results for two configurations: clsat (`r453` on SMT-EXEC), in which CLSAT is run with proof-production off; and clsat+lfsc (`r591` on SMT-EXEC), in which CLSAT is run with proof-production on, followed by a run of LFSC on the produced proof.

Table 1 Summary of results for QF_IDL (timeout: 1,800 seconds).

Configuration	Solved	Unsolved	Timeouts	Time
clsat (without proofs) (`r453`)	542	50	30	29,507.7s
clsat+lfsc (`r591`)	539	51	32	38,833.6s

The **Solved** column gives the number of benchmarks each configuration completed successfully. The **Unsolved** column gives the number of benchmarks each configuration failed to solve before timeout due to CLSAT's incomplete CNF conversion implementation and lack of arbitrary precision arithmetic. The first configuration solved all benchmarks solved by the second. The one additional unsolved answer for clsat+lfsc is `diamonds.18.10.i.a.u.smt`, the proof of which is bigger than 2GB in size and the proof checker failed on due to a memory overflow. The **Time** column gives the total times taken by each configuration to solve the 539 benchmarks solved by both. Those totals show that the overall overhead of proof generation *and* proof checking over just solving for those benchmarks was 31.6%, which we consider rather reasonable.

For a more detailed picture on the overhead incurred with proof-checking, Figure 17 compares proof checking times by clsat+lfsc with CLSAT's solve-only times. Each dot represents one of the 542 benchmarks that CLSAT could solve. The horizontal axis is for solve-only times (without proof production) while the vertical axis is for proof checking times. Both are in seconds on a log scale. It turned out that the proofs from certain families of benchmarks, namely `fischer`, `diamonds`, `planning` and `post_office`, are much more difficult to verify than those

[8] The hardest QF_IDL benchmarks in SMT-LIB have difficulty 5.

[9] The results are publicly available at `http://www.smt-exec.org` under the SMT-EXEC job `clsat-lfsc-2009.8`.

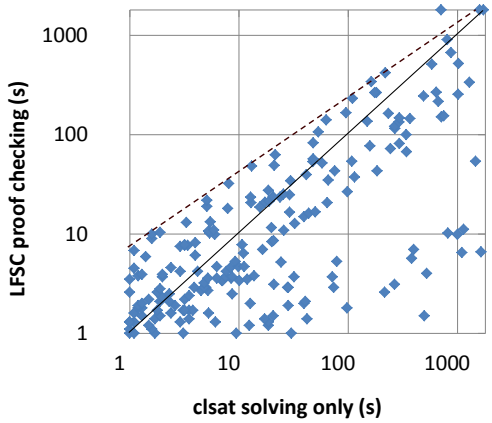

Fig. 17 Solve-only times versus proof checking times for QF_IDL.

for other families. In particular, for those benchmarks proof checking took *longer* than solving. The worst offender is benchmark `diamonds.11.3.i.a.u.smt` whose proof checking time was 2.30s vs 0.2s of solving time. However, the figure also shows that as these benchmarks get more difficult, the relative proof overheads appear to converge towards 100%, as indicated by the dotted line.[10]

6.2 Results for QF_LRA

To evaluate experimentally the LFSC proof system for QF_LRA sketched in Section 5.2, we instrumented the SMT solver CVC3 to output proofs in that system. Because CVC3 already has its own proof generation module, which is tightly integrated with the rest of the solver, we generated the LFSC proofs using CVC3's native proofs as a guide. We looked at all the QF_LRA and QF_RDL unsatisfiable benchmarks from SMT-LIB.[11].

Our experimental evaluation contains no comparisons with other proof checkers besides LFSC for lack of alternative proof-generating solvers and checkers for QF_LRA. To our knowledge, the only potential candidate was a former system developed by Ge and Barrett that used the HOL Light prover as a proof checker for CVC3 [19]. Unfortunately, that system, which was never tested on QF_LRA benchmarks and was not kept in sync with the latest developments of CVC3, breaks on most of these benchmarks. Instead, as a reference point, we compared proofs produced in our LFSC proof system against proofs translated into an alternative LFSC proof system for QF_LRA that mimics the rules contained in CVC3's native proof format. These rules are mostly declarative, with a few exceptions.[12]

[10] The outlier above the dotted line is `diamonds.18.10.i.a.u.smt`.

[11] Each of these benchmarks consists of an unsatisfiable quantifier-free LRA formula. QF_RDL is a sublogic of QF_LRA.

[12] Details of these proof systems as well as proof excerpts can be found in [39].

Benchmark		**Solve + (Pf Gen) (s)**			**Pf Size (MB)**		**Pf Check (s)**		
Family	#	cvc	lrac	lra	lrac	lra	lrac	lra	T%
check-lra	1	0.1	0.3	0.2	0.5	0.1	0.1	0.0	79%
check-rdl	1	0.1	0.1	0.1	0.0	0.0	0.0	0.0	48%
clock_synch	18	11.7	21.8	21.7	14.8	13.0	2.6	2.3	17%
gasburner	19	4.0	8.6	7.8	13.9	7.7	2.5	1.6	46%
pursuit	8	16.6	26.3	26.3	5.1	3.6	0.8	0.7	36%
sal	31	1584.8	3254.3	3239.8	537.1	472.2	275.6	269.0	6%
scheduling	8	281.8	322.9	322.1	25.1	17.8	3.7	2.9	37%
spider	35	10.2	17.4	17.4	12.7	11.1	2.3	2.4	15%
tgc	21	31.5	55.9	55.4	22.7	16.9	4.2	3.4	16%
TM	1	17.6	29.3	29.0	2.7	2.7	0.4	0.4	0%
tta_startup	25	29.7	68.4	68.5	43.6	43.2	5.4	5.6	3%
uart	9	1074.2	1391.2	1434.7	102.4	76.6	42.7	37.0	13%
windowreal	24	20.6	41.6	41.7	22.1	21.9	2.8	2.9	3%
Total	201	3082.9	5238.0	5264.6	802.8	686.9	343.2	328.1	8%

Fig. 18 Cumulative results for QF_LRA, grouped by benchmark family. Column 2 gives the numbers of benchmarks in each family. Columns 3–5 give CVC3's aggregate runtime for each of the 3 configurations. Columns 6–7 show the proof sizes for the two proof-producing configurations. Columns 8–9 show LFSC proof checking times. The last column show the average theory content of each benchmark family.

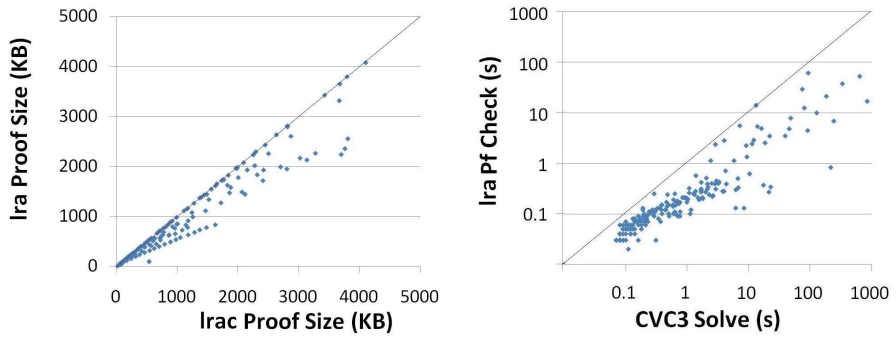

Fig. 19 Comparing proof sizes and proof checking times for QF_LRA.

We ran our experiments on a Linux machine with two 2.67GHz 4-core Xeon processors and 8GB of RAM. We discuss benchmarks for which CVC3 could generate a proof within a timeout of 900 seconds: 161 of the 317 unsatisfiable QF_LRA benchmarks, and 40 of the 113 unsatisfiable QF_RDL benchmarks. We collected runtimes for the following three main configurations of CVC3.

- cvc: Default, solving benchmarks but with no proof generation.
- lrac: Solving with proof generation in the LFSC encoding of CVC3's format.
- lra: Solving with proof generation in our LFSC proof system.

Recall that the main idea of our proof system for QF_LRA is to use a polynomial representation of LRA terms so that theory reasoning is justified with rules like those in Figure 14. Propositional reasoning steps (after CNF conversion) are encoded with the deferred resolution rule presented in Section 4.2. As a consequence, one should expect the effectiveness of the polynomial representation in reducing proof sizes and checking times to be correlated with the amount of *the-*

ory content of a proof. Concretely, we measure that as the percentage of nodes in a native CVC3 proof that belong to the (sub)proof of a theory lemma. For our benchmarks set, the average theory content was very low, about 8.3%, considerably diluting the global impact of our polynomial representation. However, this impact is clearly significant, and positive, on proofs or subproofs with high theory content, as discussed below.

Table 18 shows a summary of our results for various families of benchmarks. Since translating CVC3's native proofs into LFSC format increased proof generation time and proof sizes only by a small constant factor, we do not include these values in the table. As the table shows, CVC3's solving times (i.e., the runtimes of the cvc configuration) are on average 1.65 times faster than solving with native proof generation (the lrac configuration). The translation to proofs in our system (the lra configuration) adds additional overhead, which is however less than 3% on average.

The scatter plots in Figure 19 are helpful in comparing proof sizes and proof checking times for the two proof systems. The first plot shows that ours, lra, achieves constant size compression factors over the LFSC encoding of native CVC3 proofs, lrac. A number of benchmarks in our test set do not benefit from using our proof system. Such benchmarks are minimally dependent on theory reasoning, having a theory content of less than 2%. In contrast, for benchmarks with higher theory content, lra is effective at proof compression compared to lrac. For instance, over the set of all benchmarks with a theory content of 10% or more, proofs in our system occupy on average 24% less space than CVC3 native proofs in LFSC format. When focusing just on subproofs of theory lemmas, the average compression goes up significantly, to 81.3%; that is to say, theory lemma subproofs in our proof system are 5.3 times smaller than native CVC3 proofs of the same lemmas. Interestingly, the compression factor is not the same for all benchmarks, although an analysis of the individual results shows that benchmarks in the same SMT-LIB family tend to have the same compression factor.

It is generally expected that proof checking should be substantially faster than proof generation or even just solving. This was generally the case in our experiments for both proof systems when proof checking used compiled side conditions.[13] LFSC's proof checking times for both proof systems were around 9 times smaller than CVC3's solving times. Furthermore, checking lra proofs was always more efficient than checking the LFSC encoding of CVC3's native proofs; in particular, it was on average 2.3 times faster for proofs of theory lemmas.

Overall, our experiments with multiple LFSC proof systems for the same logic (QF_LRA), show that mixing declarative proof rules, with no side conditions, with more computational arithmetic proof rules, with fairly simple side conditions, is effective in producing smaller proof sizes and proof checking times.

7 Further applications: leveraging type inference

To illustrate the power and flexibility of the LFSC framework further, we sketch an additional research direction we have started to explore recently. More details can be found in [40]. Its starting point is the use of the wildcard symbol _ in proof terms, which requires the LFSC proof checker to perform type *inference* as opposed

[13] The LFSC checker can be used either with compiled or with interpreted side condition code.

```
(declare color type)      (declare formula type)
(declare A color)         (declare colored (! phi formula (! col color type)))
(declare B color)         (declare p_interpolant (! c clause (! phi formula type)))
(declare clause type)     (declare interpolant (! phi formula type))
```

Fig. 20 Basic definitions for encoding interpolating calculi in LFSC.

to mere type checking. One can exploit this feature by designing a logical system in which certain terms of interest in a proof need not be specified beforehand, and are instead computed as a side effect of proof checking. An immediate application can be seen in *interpolant-generating proofs*.

Given a logical theory T and two sets of formulas A and B that are jointly unsatisfiable in T, a *T-interpolant for A and B* is a formula ϕ over the symbols of T and the free symbols common to A and B such that (i) $A \models_T \phi$ and (ii) $B, \phi \models_T \bot$, where $\models_T$ is logical entailment in T and $\bot$ is the universally false formula. For certain theories, interpolants can be generated efficiently from a refutation of $A \wedge B$. Interpolants have been used successfully in a variety of contexts, including symbolic model checking [30] and predicate abstraction [23]. In many applications, it is critical that formulas computed by interpolant-generation procedures are indeed interpolants; that is, exhibit the defining properties above. LFSC offers a way of addressing this need by generating *certified interpolants*.

Extracting interpolants from refutation proofs typically involves the use of *interpolant-generating calculi*, in which inference rules are augmented with additional information necessary to construct an interpolant. For example, an interpolant for an unsatisfiable pair of propositional clause sets (A, B) can be constructed from a refutation of $A \cup B$ written in a calculus with rules based on sequents of the form $(A, B) \vdash c[\phi]$, where c is a clause and ϕ a formula. Sequents like these are commonly referred to as *partial interpolants*. When c is the empty clause, ϕ is an interpolant for (A, B). The LF language allows one to augment proof rules to carry additional information through the use of suitably modified types. This makes encoding partial interpolants into LFSC straightforward. For the example above, the dependent type (holds c) used in Section 4.1 can be replaced with (p_interpolant c ϕ), where again c is a clause and ϕ a formula annotating c. This general scheme may be used when devising encodings of interpolant generating calculi for theories as well.

Figure 20 provides some basic definitions common to the encodings of interpolant generating calculi in LFSC for two sets A and B of input formulas. We use a base type color with two nullary constructors, A and B. Colors are used as a way to tag an input formula ψ as occurring in the set A or B, through the type (colored ϕ *col*), where *col* is either A or B. Since formulas in the sets A and B can be inferred from the types of the free variables in proof terms, the sets do not need to be explicitly recorded as part of proof judgment types. In particular, our judgment for interpolants is encoded as the type (interpolant ϕ).

Our approach allows for two options to obtain certified interpolants. With the first, an LFSC proof term P can be checked against the type (interpolant ϕ) for some *given* formula ϕ. In other words, the alleged interpolant ϕ is explicitly provided as part of the proof, and if proof checking succeeds, then both the proof P and the interpolant ϕ are certified to be correct. Note that in this case the user and the proof checker must agree on the exact form of the interpolant ϕ. Alternatively,

P can be checked against the type schema (interpolant _). If the proof checker verifies that P has type (interpolant ϕ) for some formula ϕ generated by type inference, it will output ϕ. In this case, interpolant generation comes as a side effect of proof checking, and the returned interpolant ϕ is correct by construction.

In recent experimental work [40] we found that interpolants can be generated using LFSC with a small overhead with respect to solving. In that work, we focused on interpolant generation for the theory of equality and uninterpreted functions (EUF), using the LFSC proof checker as an interpolant generator for proofs generated by CVC3. A simple calculus for interpolant generation in EUF can be encoded in LFSC in a natural way, with minimal dependence upon side conditions. Overall, our experiments showed that interpolant generation had a 22% overhead with respect to solving with proof generation, indicating that the generation of certified interpolants is practicably feasible with high-performance SMT solvers.

8 Conclusion and future work

We have argued how efficient and highly customizable proof checking can be supported with the Logical Framework with Side Conditions, LFSC. We have shown how diverse proof rules, from purely propositional to core SMT to theory inferences, can be supported naturally and efficiently using LFSC. Thanks to an optimized implementation, LFSC proof checking times compare very favorably to solving times, using two independently developed SMT solvers. We have also shown how more advanced applications, such as interpolant generation, can also be supported by LFSC. This illustrates the further benefits of using a framework beyond basic proof checking.

Future work includes a new, more user-friendly syntax for LFSC signatures, a simpler and leaner, from scratch re-implementation of the LFSC checker—intended for public release and under way—as well as additional theories and applications of LFSC for SMT. We also intend to pursue the ultimate goal of a standard SMT-LIB proof format, based on ideas from LFSC, as well as the broader SMT community.

Acknowledgements We would like to thank Yeting Ge and Clark Barrett for their help translating CVC3's proof format into LFSC. We also thank the anonymous referees for their thoughtful and detailed reviews whose suggestions have considerably helped us improve the presentation.

References

1. A. Armando, J.M., Platania., L.: Bounded model checking of software using SMT solvers instead of SAT solvers. In: Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'06), *Lecture Notes in Computer Science*, vol. 3925, pp. 146–162. Springer (2006)
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: J.P. Jouannaud, Z. Shao (eds.) Certified Programs and Proofs, *Lecture Notes in Computer Science*, vol. 7086, pp. 135–150. Springer (2011)
3. Barnett, M., yuh Evan Chang, B., Deline, R., Jacobs, B., Leino, K.R.: Boogie: A modular reusable verifier for object-oriented programs. In: 4th International Symposium on Formal Methods for Components and Objects, *Lecture Notes in Computer Science*, vol. 4111, pp. 364–387. Springer (2006)

4. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: A. Biere, M.J.H. Heule, H. van Maaren, T. Walsh (eds.) Handbook of Satisfiability, vol. 185, chap. 26, pp. 825–885. IOS Press (2009)
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: A. Gupta, D. Kroening (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England) (2010). Available from `www.smtlib.org`
6. Barrett, C., Tinelli, C.: CVC3. In: W. Damm, H. Hermanns (eds.) Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany, *Lecture Notes in Computer Science*, vol. 4590, pp. 298–302. Springer (2007)
7. Bauer, L., Garriss, S., McCune, J.M., Reiter, M.K., Rouse, J., Rutenbar, P.: Device-enabled authorization in the Grey system. In: Proceedings of the 8th Information Security Conference (ISC'05), pp. 431–445 (2005)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
9. Besson, F., Fontaine, P., Théry, L.: A Flexible Proof Format for SMT: a Proposal. In: P. Fontaine, A. Stump (eds.) Workshop on Proof eXchange for Theorem Proving (PxTP) (2011)
10. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. Wrocław, Poland (2011)
11. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: M. Kaufmann, L. Paulson (eds.) Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 6172, pp. 179–194. Springer (2010)
12. Bouton, T., Caminha B. De Oliveira, D., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: R.A. Schmidt (ed.) Proceedings of the 22nd International Conference on Automated Deduction (CADE), CADE-22, pp. 151–156. Springer-Verlag (2009)
13. Chen, J., Chugh, R., Swamy, N.: Type-preserving compilation of end-to-end verification of security enforcement. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 412–423. ACM (2010)
14. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1), 7–34 (2001)
15. Deharbe, D., Fontaine, P., Paleo, B.W.: Quantifier inference rules for SMT proofs. In: Workshop on Proof eXchange for Theorem Proving (2011)
16. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B.: Extended static checking for Java. In: Proc. ACM Conference on Programming Language Design and Implementation, pp. 234–245 (2002)
17. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: In Tools and Algorithms for Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 3920, pp. 167–181. Springer-Verlag (2006)
18. Ford, J., Shankar, N.: Formal verification of a combination decision procedure. In: A. Voronkov (ed.) 18th International Conference on Automated Deduction (CADE), *Lecture Notes in Computer Science*, vol. 2392, pp. 347–362. Springer (2002)
19. Ge, Y., Barrett, C.: Proof translation and SMT-LIB benchmark certification: A preliminary report. In: Proceedings of International Workshop on Satisfiability Modulo Theories (2008)
20. Goel, A., Krstić, S., Tinelli, C.: Ground interpolation for combined theories. In: R. Schmidt (ed.) Proceedings of the 22nd International Conference on Automated Deduction (CADE), *Lecture Notes in Artificial Intelligence*, vol. 5663, pp. 183–198. Springer (2009)
21. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: A. Cimatti, R. Jones (eds.) Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon), pp. 109–117. IEEE (2008)
22. Harper, R., Honsell, F., Plotkin, G.: A Framework for Defining Logics. Journal of the Association for Computing Machinery **40**(1), 143–184 (1993)
23. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 3920, pp. 459–473. Springer (2006)

24. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: J. Matthews, T. Anderson (eds.) 22nd ACM Symposium on Operating Systems Principles (SOSP), pp. 207–220. ACM (2009)
25. Kothari, N., Mahajan, R., Millstein, T.D., Govindan, R., Musuvathi, M.: Finding protocol manipulation attacks. In: S. Keshav, J. Liebeherr, J.W. Byers, J.C. Mogul (eds.) Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 26–37 (2011)
26. Lee, D., Crary, K., Harper, R.: Towards a Mechanized Metatheory of Standard ML. In: Proceedings of 34th ACM Symposium on Principles of Programming Languages, pp. 173–184. ACM Press (2007)
27. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: G. Morrisett, S.P. Jones (eds.) 33rd ACM symposium on Principles of Programming Languages, pp. 42–54. ACM Press (2006)
28. Lescuyer, S., Conchon, S.: 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)
29. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theoretical Computer Science **411**, 4333–4356 (2010)
30. McMillan, K.: Interpolation and SAT-based model checking. In: W.A.H. Jr., F. Somenzi (eds.) Proceedings of Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 2725, pp. 1–13. Springer (2003)
31. Moskal, M.: Rocket-Fast Proof Checking for SMT Solvers. In: C. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 4963, pp. 486–500. Springer (2008)
32. de Moura, L., Bjørner, N.: Proofs and Refutations, and Z3. In: B. Konev, R. Schmidt, S. Schulz (eds.) 7th International Workshop on the Implementation of Logics (IWIL) (2008)
33. Necula, G.: Proof-Carrying Code. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106–119 (1997)
34. Necula, G., Lee, P.: Efficient representation and validation of proofs. In: 13th Annual IEEE Symposium on Logic in Computer Science, pp. 93–104 (1998)
35. Necula, G., Rahul, S.: Oracle-Based Checking of Untrusted Software. In: Proceedings of the 28th ACM Symposium on Principles of Programming Languages, pp. 142–154 (2001)
36. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM **53**(6), 937–977 (2006)
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer (2002)
38. Oe, D., Reynolds, A., Stump, A.: Fast and Flexible Proof Checking for SMT. In: B. Dutertre, O. Strichman (eds.) Proceedings of International Workshop on Satisfiability Modulo Theories (2009)
39. Reynolds, A., Hadarean, L., Tinelli, C., Ge, Y., Stump, A., Barrett, C.: Comparing proof systems for linear real arithmetic with LFSC. In: A. Gupta, D. Kroening (eds.) Proceedings of International Workshop on Satisfiability Modulo Theories (2010)
40. Reynolds, A., Tinelli, C., Hadarean, L.: Certified interpolant generation for EUF. In: S. Lahiri, S. Seshia (eds.) Proceedings of the 9th International Workshop on Satisfiability Modulo Theories (2011)
41. Robinson, J., Voronkov, A.E.: Handbook of Automated Reasoning. Elsevier Science Publishers and MIT Press (2001)
42. Sebastiani, R.: Lazy satisability modulo theories. Journal on Satisfiability, Boolean Modeling and Computation **3**(3-4), 141–224 (2007)
43. Stump, A.: Proof checking technology for satisfiability modulo theories. In: A. Abel, C. Urban (eds.) Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP) (2008)
44. Stump, A., Dill, D.: Faster Proof Checking in the Edinburgh Logical Framework. In: 18th International Conference on Automated Deduction (CADE), pp. 392–407 (2002)
45. Stump, A., Oe, D.: Towards an SMT Proof Format. In: C. Barrett, L. de Moura (eds.) Proceedings of International Workshop on Satisfiability Modulo Theories (2008)
46. Van Gelder, A.: URL `http://users.soe.ucsc.edu/~avg/ProofChecker/ProofChecker-fileformat.txt`

47. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A Concurrent Logical Framework I: Judgments and Properties. Tech. Rep. CMU-CS-02-101, Carnegie Mellon University (2002)
48. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic **7**(1), 26 – 40 (2009)
49. Zee, K., Kuncak, V., Rinard, M.C.: An integrated proof language for imperative programs. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 338–351 (2009)
50. Zeller, M., Stump, A., Deters, M.: Signature Compilation for the Edinburgh Logical Framework. In: C. Schürmann (ed.) Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP) (2007)
51. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: Proceedings of 8th International Conference on Computer Aided Deduction (CADE) (2002)

A Typing Rules for Code

The completely standard type-computation rules for code terms are given in Figure 21. Code terms are monomorphically typed. 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, $\mathtt{ctxt}(P) = \{x_1 : T_1, \ldots x_n : T_n\}$, where c has type $\Pi x_1 : T_1. \cdots x_n : T_n.T$. We sometimes write $(\mathtt{do}\ C_1\ C_2)$ as an abbreviation for $(\mathtt{let}\ x\ C_1\ C_2)$, where $x \notin \mathtt{FV}(C_2)$.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow T} \qquad \frac{}{\Gamma \vdash N \Rightarrow \mathtt{mpz}}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow \mathtt{mpz} \quad \Gamma \vdash t_2 \Rightarrow \mathtt{mpz}}{\Gamma \vdash t_1 + t_2 \Rightarrow \mathtt{mpz}} \qquad \frac{\Gamma \vdash t \Rightarrow \mathtt{mpz}}{\Gamma \vdash -t \Rightarrow \mathtt{mpz}}$$

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

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

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

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

Fig. 21 Typing Rules for Code Terms. Rules for the built-in rational type are similar to those for the the integer type, and so are omitted.

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 22 and 23. We can note the frequent uses of match, for decomposing or testing the form of data. The program eqvar of Figure 22 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,

```
(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))))))
```

Fig. 22 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))))))))
```

Fig. 23 Operations on clauses.

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 Small Example Proof

Figure 24 shows a small QF_IDL proof. This proof derives a contradiction from the assumed formula

```
(and (<= (- x y) (as_int (~ 1)))
```

```
(% x int (% y int (% z int
(% f (th_holds (and (<= (- x y) (as_int (~ 1)))
                    (and (<= (- y z) (as_int (~ 2)))
                         (<= (- z x) (as_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
  (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)
))))))))))))))))))))))))))))))))
```

Fig. 24 A small QF_IDL proof

```
                    (and (<= (- y z) (as_int (~ 2)))
                         (<= (- z x) (as_int (~ 3)))))
```

The proof begins by introducing the variables `x`, `y`, and `z`, and the assumption (named `f`) of the formula above. Then it uses CNF conversion rules to put that formula into CNF. CNF conversion starts with an application of the start rule, which turns the hypothesis of the input formula (th_hold ϕ) to a proof of the partial clause (pc_hold (ϕ;)). The `dist_pos`, mentioned also in Section 4.3 above, breaks a conjunctive partial clause into conjuncts. The `decl_atom_pos` proof rule introduces new propositional variables for positive occurrences of atomic formulas. The new propositional variables introduced this way are `v0`, `v1`, and `v2`, corresponding to the atomic formulas (let us call them ϕ_0, ϕ_1, and ϕ_2) in the original assumed formula, in order. The `decl_atom_pos` rule is similar to rename (discussed in Section 4.3 above), but it also binds additional meta-variables of type (atom v ϕ) to record the relationships between variables and abstracted formulas. So for example, `a0` is of type `(atom v0` ϕ_0`)`. The clausify rule turns the judgements of partial clauses with the empty formula sequence (pc_holds (; c)) to the judgements of pure propositional clauses (holds c). Here, this introduces variables `x0`, `x1`, and `x2` as names for the asserted unit clauses ϕ_0, ϕ_1, and ϕ_2, respectively.

After CNF conversion is complete, the proof derives a contradiction from those asserted unit clauses and a theory clause derived using `assume_true` (see Section 4.4) from a theory contradiction. The theory contradiction is obtained with `idl_trans` and `idl_contra` (Section 5.1).