From Declarative to Computational Proof Checking for LRA

Andrew Reynolds[1], Liana Hadarean[2], Cesare Tinelli[1],
Yeting Ge[2], Aaron Stump[1], and Clark Barrett[2]

[1]The University of Iowa [2]New York University

Abstract. In this work, we investigate various proof systems for quantifier-free Linear Real Arithmetic, focusing on the continuum between declarative and computational styles of proof checking. We use LFSC, a high-level declarative language for defining proof systems and proof objects for virtually any logic. One of the distinguishing features of LFSC is its support for computational side conditions on proof rules. Side conditions facilitate the design of proof systems that reflect closely the sort of high-performance inferences made by SMT solvers. We propose a proof translation for LRA that exploits the continuum between declarative and computational proof checking, and report on our comparative experimental results on generating and checking proofs using alternative strategies.

1 Introduction

Automated verification techniques increasingly rely on Satisfiability Modulo Theories (SMT) solvers to discharge verification conditions (e.g., [5, 10, 16]). The ability for an SMT solver to produce a proof when it reports a formula unsatisfiable has long been recognized as valuable, and is increasingly important for applications. For verification of safety-critical systems, for example, it is important to be able to trust (and indeed possibly certify) the results of the solver. The complexity of modern SMT solvers makes verifying the solver itself a formidable challenge. Producing and checking a proof with a much simpler and more trustworthy, possibly even certified, proof checker is an attractive alternative. Furthermore, some applications (e.g., [6, 5]) make use of proofs produced by SMT solvers, so even if a solver were completely trusted, there would still be a need for proof production.

The diversity of theories and solving algorithms within SMT makes establishing a common SMT proof format difficult, as it seems practically infeasible to design a single set of universally suitable inference rules. To address this difficulty, previous work introduced LFSC ("Logical Framework with Side Conditions"), a meta-language for specifying proof systems in SMT [17], and showed how to apply it for encoding proofs in the quantifier-free integer difference logic (QF_IDL) [14] of SMT-LIB [1]. LFSC is based on the Edinburgh Logical Framework (LF), a high-level declarative language in which it is possible to specify logics [8]. LFSC increases LF's flexibility by including support for computational side conditions on inference rules. 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.

As with LF, with LFSC a *single* high-performance type checker can check proofs in any LFSC-specified logic. The presence of side conditions in LFSC opens up a continuum of possible LFSC encodings of a given inference system, from completely declarative, using rules with no side conditions, at one end; to completely computational, using a single rule with a huge side condition, at the other. Solver implementers thus have the freedom to choose how to divide a proof system into declarative and computational parts. Such design decisions are explicitly recorded in the LFSC specification and become part of the trusted computing base.

The advantages of tools using interactive theorem provers such as Isabelle/HOL are well known; in particular, their trusted core contains only a base logic and a small fixed number of proof rules. While recent work [4] has significantly improved checking times for LCF-style proofs, the performance of these provers still lags behind C/C++ checkers carefully engineered for fast checking of very large proofs. The approach of LFSC seeks to strike a pragmatic compromise between trustworthiness and efficiency. It would certainly be possible to reduce the size and complexity of the trusted computing base of the LFSC type checker by using a less optimized implementation, at the cost of reduced performance.

In this work, we use LFSC to investigate alternative translation methods for proofs generated for linear real arithmetic (LRA). Conventional approaches to proof generation in SMT often involve the use of declarative rewrite steps, which as a whole, represent a trace of the arithmetic inferences applied by the prover. While this level of granularity makes the design of a proof system straightforward, it does present practical concerns, such as overall proof size. For LRA, it is widely known that a certificate of unsatisfiability for a system of inequations can be concisely represented as a set of coefficients which, when multiplied by the inequations and summed together give an trivially inconsistent inequation (for instance, one of the form $c \leq 0$ for some positive constant c). Depending on the decision procedure used, determining these coefficients a posteriori can be a non-trivial task. We describe a generalized proof translation in which these coefficients are efficiently computed from declarative *trace-style* proofs, making use of LFSC as a meta-framework for defining proof rules involving computational side conditions.

Contributions. We present a novel translation from the declarative style proofs produced by the Cvc3 SMT solver [2] to an alternative, coefficient-based proof system, taking advantage of LFSC's side conditions. First we provide an LFSC formalization of Cvc3's proof system for the quantifier-free fragment of Linear Real Arithmetic (QF_LRA) [1], and instrument Cvc3 to produce proofs in LFSC format. That formalization mainly uses declarative rules with no side conditions, many of which represent explicit rewrite steps. Then we provide a translation to LFSC proofs that use novel, coefficient-based proof rules relying on side conditions. In our experimental evaluation, the translated proofs of arithmetic lemmas (valid disjunctions of literals) are on average 5.3 times more compact and 2.3 times faster to check than the original Cvc3 proofs.

Paper outline. We begin with a brief introduction to LFSC. In Section 3, we describe abstractly the LFSC LRA calculus, as well as Cvc3's, in terms of textbook logic rules.

Then we explain how proofs in the native format of Cvc3 are translated in LFSC format, and subsequently converted to coefficient-based proofs. We include a formal definition of the translation applied to a subset of Cvc3 proof rules. In Section 4, we provide comparative experimental results on generating and checking LFSC proofs using various translations. Finally, we conclude with a few notes on further work.

2 LF with Side Conditions

LFSC extends the Edinburgh LF with support for computational side conditions. LF has been used extensively as a metalanguage for encoding deductive systems including logics, semantics of programming languages, as well as many other applications [9, 3, 12]. LF is a type-theoretic language, where proof systems are encoded as *signatures*, collections typing declarations. Each proof rule is encoded as a constant symbol, whose type represents the inference allowed by the rule. For example, the following transitivity rule for equality

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

can be encoded in LF (using the prefix syntax of LFSC) as

```
eq_trans :  (! t1 term (! t2 term (! t3 term
              (! u1 (pf (= t1 t2)) (! u2 (pf (= t2 t3))
                (pf (= t1 t3)))))))
```

where `!` represents LF's Π binder, for the dependent function space. The encoded rule can be understood intuitively as saying: "for any terms t_1, t_2 and t_3, and any proofs u_1 and u_2 of $t_1 = t_2$ and $t_2 = t_3$ respectively, `eq_trans` constructs a proof of $t_1 = t_3$."

Pure LF is not suitable for encoding large proofs from SMT solvers, due to the computational nature of many SMT inferences. For example, consider trying to prove the following seemingly immediate statement:

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

where $t_{i_1} \ldots t_{i_n}$ is a permutation of the terms $t_1, \ldots, t_n$. A purely declarative proof would (seem to) require sorting the terms using associativity and commutativity of binary $+$, thus requiring $\Omega(n \log n)$ applications of proof rules encoding those properties. To circumvent this problem, LFSC allows rules with side conditions expressed as *computational checks* written in a strongly typed first-order language with a simple form of ML-style pattern matching.

When checking the application of an inference rule with a side condition, the LFSC checker computes actual parameters for the side condition and executes its code. If the side condition fails, either because it is not satisfied or because of an exception caused by a pattern-matching failure, the LFSC checker rejects the application of the rule. Thus, statement (1) could be proven using a single LFSC inference rule of the form:

```
eq_zero :  (! t term
             (! sc (^ (simplify t) 0)
               (pf (t = 0))))
```

where `simplify` 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 `(^ (simplify t) 0)` defines the side condition of the `eq_zero` rule, with the condition succeeding iff the expression `(simplify t)` evaluates to `0`. The `simplify` function might have a definition like the following (where `tt` and `ff` denote constructors for boolean true and false respectively):

```
(program simplify ((t term)) term
  (match (normalize t)
    ((poly c l)
      (match (is_zero l)
        (tt c)
        (ff fail)))))
```

where `normalize` and `is_zero` are auxiliary functions defined similarly. Here, polynomials are represented as (inductive data type) values of the form `(poly` c l`)` where c is the constant part of the polynomial and l is the rest. This side condition will first convert term `t` to a polynomial, and then attempt to simplify the non-constant part of the result to zero. If successful, it will return the constant part `c` of the polynomial.

The `eq_zero` rule is a simple example of the opposite end of the LFSC spectrum, i.e., fully computational proof checking. Many of the proof inferences described in the rest of the paper fall somewhere in-between declarative and computational, with terms being normalized to polynomials and side condition functions performing mostly operations over such polynomials. These functions are comparable in size and complexity to the function `simplify` above.

Proof Checking. LFSC proofs are checked using a high-performance LFSC type checker, developed (in around 5kloc of C++) by Reynolds and Stump [14]. The checker takes as input a signature and a proof in that signature to be checked. The signatures discussed in this paper contain about 60 lines of side condition code, for a total of less than 2 kilobytes. As a whole, this code is similar in structural complexity to the implementation of a merge sort of key/value pairs in LISP.

As described in [14], the LFSC checker compiles side-condition code to C++, for higher performance. Moreover, it performs *incremental checking*, interleaving parsing and type checking. This allows it to parse and type check large proofs, without needing to build first an abstract syntax tree in memory for the whole proof. These two optimizations each lead to significant reductions in running time (on the order of 5x) and memory usage.

While LFSC is rather general and fairly stable, it is still under active development and may occasionally require the addition of new features for better support. With this work, a few new features were added to the LFSC language and type checker: (i) support for arbitrary-precision rational arithmetic; (ii) local definitions via a `let` construct; and (iii) a primitive `compare` function imposing a total ordering on LFSC variables, which enables sorting in the side condition language.

3 Proof Generation and Checking for LRA

We formalize the quantifier-free fragment of LRA in an LFSC proof system (or *calculus*), which we call $\mathcal{L}$ here. Proofs in the $\mathcal{L}$ calculus are generated from proofs produced by Cvc3 in its own calculus. Since Cvc3's proof-generation facility is deeply embedded in the system's code, a translation module was added to Cvc3 that traverses the internal data structure storing the proof, and produces an LFSC proof from it.

Roughly speaking, Cvc3's proofs have a two-tiered structure, typical of solvers based on the DPLL(T) architecture [13], with a propositional skeleton filled with several theory-specific subproofs. The proof's conclusion is reached by means of propositional or purely equational inferences applied to a set of input formulas and a set of *theory lemmas*. The latter are disjunctions of arithmetic atoms deduced from no assumptions, mostly using proof rules specific to the theory in question—the theory of real arithmetic in this case.

We implemented two different translations from Cvc3 proofs, described below, differing in how close they are to the original proof. We will refer to these as the *literal* and the *liberal* translation, and name them Lit and Lib, respectively.

Literal translation. In Lit, a trace-style LFSC proof is produced directly from Cvc3's proof, using whenever possible declarative $\mathcal{L}$ rules that mirror the corresponding Cvc3 rules, and resorting to $\mathcal{L}$-specific rules involving side conditions only for those few Cvc3 rules that cannot be checked by simple pattern matching (but require, for instance, to verify that a certain expression in the Cvc3 rule is the canonical version of another). The set of rules used in this translation is in effect a subcalculus of $\mathcal{L}$. We will refer to it as the $\mathcal{C}$ calculus.

Liberal translation. In Lib, the Cvc3 proof is used as a guide to produce a compact coefficient-based proof that relies on rules with side conditions. The use of side conditions enables compaction that is otherwise infeasible due to the declarative nature of the Cvc3 rules. In Lib, the subproofs of all theory lemmas are systematically converted to more compact proofs that use $\mathcal{L}$-specific rules. The rest of the Cvc3 proof—which does not involve LRA-specific reasoning—is translated in the same way as in the literal translation.

We also experimented with a third translation, a more aggressive version of Lib, which also tries to compact portions of the Cvc3 proof that rely on general equality reasoning. This translation uses an adaptive strategy to switch from $\mathcal{L}$-specific equality rules to $\mathcal{C}$ equality rules and back, making heuristic decisions on when it is worthwhile to do so. Although not discussed further here, experiments on this translation are included in a companion report [15], showing mixed results for different classes of benchmarks.

The proof translation times for the Lit and the Lib translations are roughly the same. However, the size of a proof produced with the liberal translation is often considerably smaller than the size of the corresponding proof generated with the literal translation. The compression is achieved to a great extent thanks to $\mathcal{L}$ proof rules that work with *normalized polynomial atoms*, atoms of the form

$$c_1 \cdot x_1 + \ldots + c_n \cdot x_n + c_{n+1} \sim 0$$

$$\frac{\varphi_1 \Leftrightarrow \varphi_2 \quad \varphi_2 \Leftrightarrow \varphi_3}{\varphi_1 \Leftrightarrow \varphi_3}\ \textsf{iff\_trans} \qquad \frac{\varphi_1 \quad \varphi_1 \Leftrightarrow \varphi_2}{\varphi_2}\ \textsf{iff\_mp}$$

$$\frac{t_1 = t_2 \quad t_3 = t_4}{t_1 \sim t_3 \Leftrightarrow t_2 \sim t_4}\ \textsf{congr\_1} \qquad \frac{t_1 = t_2 \quad t_2 = t_3}{t_1 \ = t_3}\ \textsf{eq\_trans}$$

$$\frac{t_1 = t_2 \quad t_3 = t_4}{t_1 \bowtie t_3 = t_2 \bowtie t_4}\ \textsf{congr\_2} \qquad \frac{t_1 = t_2}{t_2 = t_1}\ \textsf{eq\_symm}$$

$$\frac{t_1 > t_2 \quad t_2 > t_3}{t_1 > t_3}\ \textsf{gt\_trans} \qquad \frac{t_1 > t_2 \quad t_2 > t_1}{\bot}\ \textsf{gt\_acyc}$$

$$\frac{\{0 \not\sim c\}}{(0 \sim c) \Leftrightarrow \bot}\ \textsf{const\_pred\_1} \qquad \frac{}{t_1 \sim t_2 \Leftrightarrow 0 \sim t_2 - t_1}\ \textsf{right\_minus\_left}$$

$$\frac{\{t' \text{ canonical form of } t\}}{t = t'}\ \textsf{canon} \qquad \frac{\{c \text{ non-zero}\}}{t_1 = t_2 \Leftrightarrow c \cdot t_1 = c \cdot t_2}\ \textsf{mult\_eqn}$$

$$\frac{}{t_1 > t_2 \Leftrightarrow t_2 < t_1}\ \textsf{flip\_ineq} \qquad \frac{}{t_1 \sim t_2 \Leftrightarrow t_1 + t_3 \sim t_2 + t_3}\ \textsf{plus\_pred}$$

Fig. 1. Some of Cvc3's proof rules for QF_LRA. The letters φ, t and c respectively denote arithmetic formulas, terms and (concrete) constants, while $\bowtie$ denotes one of $+, -, \cdot$.

where each c_i is a rational constant, each x_i is a real variable, and $\sim$ is one of the operators $=, >, \geq$. These rules take normalized atoms as premises and produce only normalized atoms as their conclusion. The computation of these atoms is delegated to the rule's side condition, which will perform arithmetic operations on polynomials, each producing a normal form for the conclusion atom.

The rest of this section provides an abstract overview of the $\mathcal{C}$ and $\mathcal{L}$ calculi, and describes the liberal translation from Cvc3 proofs to LFSC proofs in $\mathcal{L}$. For convenience, thanks to the literal translation, we will identify Cvc3 native proofs with LFSC proofs in $\mathcal{C}$, and define the liberal translation formally as a mapping from $\mathcal{C}$-proofs to $\mathcal{L}$-proofs. Full details on the two calculi, the translations and their correctness can be found in the companion report [15].

The $\mathcal{C}$ calculus. In essence, the $\mathcal{C}$ calculus is a natural deduction-style version of a suitable fragment of the sequent-style calculus used by Cvc3.[1] Proofs in $\mathcal{C}$ derive a quantifier-free LRA formula φ from a set Γ of premises, all of which are also quantifier-free LRA formulas. A sample of $\mathcal{C}$ rules directly corresponding to Cvc3's rules is provided in Figure 1.[2] The rules are fairly standard and self-explanatory, with the possible exception of canon, which asserts an equality between a term t and its equivalent *canonical form* produced by Cvc3's canonizer module.

As a whole, the Cvc3-corresponding rules are used to represent a trace of the reasoning used by Cvc3's decision procedure for QF_LRA. This procedure relies heavily

[1] The entire Cvc3 calculus is a lot bigger because Cvc3 supports a much larger logic than QF_LRA.

[2] To ease formatting, some rules may have a different name from the corresponding rules in Cvc3.

$$\frac{p = 0}{(c \cdot p){\downarrow} = 0}\ \textsf{lra\_mult\_c=} \qquad \frac{p > 0 \quad \{c > 0\}}{(c \cdot p){\downarrow} > 0}\ \textsf{lra\_mult\_c>}$$

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

$$\frac{\{c \sim 0\}}{c \sim 0}\ \textsf{lra\_axiom}{\sim} \qquad \frac{p \sim 0 \quad \{p \nsim 0\}}{\bot}\ \textsf{lra\_contra}{\sim}$$

$$\frac{p \geq 0 \quad p' \geq 0 \quad \{p + p' = 0\}}{p = 0}\ \textsf{lra}{\geq}{\geq}\textsf{to=}$$

Fig. 2. Some of the polynomial rules of $\mathcal{L}$. The letter p denotes polynomials.

on small-step rewrites of terms and formulas, many of which are explicitly recorded as applications of proof rules by Cvc3's proof generation process. This enables a proof checker to examine a fine-grained, verbatim account of the operations used in deducing a particular formula.

Although the $\mathcal{C}$ calculus itself is quite general, all Cvc3 proofs for QF_LRA are *refutations*, that is, they prove the unsatisfiable formula $\bot$ from a subset of the formulas whose joint satisfiability Cvc3 was asked to check.

The $\mathcal{L}$ calculus. The $\mathcal{L}$ calculus is a proper superset of $\mathcal{C}$. For the purposes of optimization, the liberal translation uses proof rules to convert arithmetic terms (used in $\mathcal{C}$) to polynomials. A further set of rules operate only on polynomial atoms and are used by the liberal translation to generate proofs of LRA theory lemmas. A sample of these rules is provided in Figure 2. To ease formatting, side conditions are written together with the premises, but enclosed in braces. Although side conditions use the same syntax used in the sequents, they should be read as a mathematical notation. For example, $p = 0$ in a premise denotes an atomic formula whose left-hand side is an arbitrary polynomial and whose right-hand side is the 0 polynomial; in contrast, the side condition $\{p+p' = 0\}$, say, denotes the result of checking whether the expression $p+p'$ evaluates to 0 in the polynomial ring $\mathbb{Q}[X]$, where $\mathbb{Q}$ is the field of rational numbers and X the set of all variables (or "free constants" in SMT-LIB parlance). An expression of the form $e{\downarrow}$ denotes the result of normalizing the 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.

3.1 From Cvc3 proofs to $\mathcal{L}$ proofs

As mentioned earlier, the literal translation essentially rewrites a Cvc3 proof in LFSC format, resulting in a proof in $\mathcal{C}$. Here we will focus only on the approach used by the liberal translation, which produces substantially different and more compact proofs for theory lemmas in the $\mathcal{L}$ calculus.

Theory lemmas in Cvc3 are derived by proving a contradiction from a set of theory atoms, for instance, $\{2x = 2y,\ y = x + 5\}$. Given a proof of $\bot$ from assumptions $\varphi_1, \ldots, \varphi_n$, Cvc3 will produce the lemma $\neg\varphi_1 \vee \ldots \vee \neg\varphi_n$. Such proofs rely on a variety of rules, including rewrite axioms of the form $\psi_1 \Leftrightarrow \psi_2$ and standard rules for natural deduction.

In contrast, our theory lemma proofs in the $\mathcal{L}$ calculus amount to determining a list of rational coefficients that when multiplied by the asserted atomic formulas, allow one to produce an inconsistent polynomial atom $c_p \sim 0$, for some constant polynomial c_p. For example, multiplying $2x = 2y$ and $y = x + 5$ respectively by the coefficients $-\frac{1}{2}$ and 1, adding the resulting equations together, and normalizing gives us the atom $5 = 0$. The goal of the translation is to determine these coefficients, and subsequently compute the necessary multiplications and summations on polynomials using computational side conditions. As we will see, these coefficients are computed directly and efficiently from the Cvc3 proof.

3.2 The liberal translation Lib

In the following, we give a formal abstract definition of our translation Lib for a subset[3] of $\mathcal{C}$ proof rules, starting with a formal definition of proof in the $\mathcal{L}$ calculus (which, recall, includes all the proof rules of $\mathcal{C}$).

Definition 1 (Proof). *A proof P is either an individual formula or a triple consisting (recursively) of (i) a possibly empty set of proofs, the immediate subproofs of P, (ii) the name of a rule in $\mathcal{L}$, and (iii) a formula, the conclusion of P. We say that a proof P is a well-formed proof of φ from Γ, and write $P : \Gamma \vdash \varphi$, where Γ is a set of formulas and φ a formula iff*

1. *P has the form φ with $\varphi \in \Gamma$, or*
2. *P has the form $(\{P_1, \dots P_n\}, r, \varphi)$ and*
   (a) *$P_i : \Gamma_i \vdash \varphi_i$ for some φ_i and Γ_i, for each i,*
   (b) *applying r to $\varphi_1 \dots \varphi_n$ produces φ,*
   (c) *all formulas in $\Gamma_i \setminus \Gamma$ are local assumptions of r, for each i.*

Similarly, a $\mathcal{C}$-proof is a proof of the form above where instead r is a rule of $\mathcal{C}$.

The purpose of proof checking is to verify that a given proof (in the sense above) is a well-formed proof.[4] We will write proofs P of the second form graphically as follows, where $P_1 \dots P_n$ are the direct subproofs of P:

$$\frac{P_1 : \Gamma_1 \vdash \varphi_1 \ \cdots \ P_n : \Gamma_n \vdash \varphi_n}{P : \Gamma \vdash \varphi}\ r$$

When convenient, we will omit rule names and annotations (P_i :) for unnamed subproofs. Also, we will write $P : \varphi$ as shorthand for $P : \Gamma \vdash \varphi$ when Γ is understood or not important.

To each conclusion φ of a $\mathcal{C}$ proof (e.g., $2x + y = 2y$) we associate a unique polynomial atom $p \sim 0$ logically equivalent to φ, i.e., satisfied by the same variable

[3] See [15] for details of Lib applied to all proof rules.

[4] At the concrete, LFSC level, this translates into checking that the term encoding a proof is well-typed in the signature encoding the calculus.

assignments that satisfy φ (e.g., $(2x-y)\!\downarrow = 0$). We will denote such logical equivalence by writing $p \sim 0 \equiv \varphi$ (with $\equiv$ not to be confused with the object syntax $\Leftrightarrow$ used for double implication).

Let $\mathbf{P}$ be the set of all proofs. In [15], we define a set $\mathbf{P}_{\text{lra}} \subseteq \mathbf{P}$ of *theory reasoning $\mathcal{C}$-proofs* and a total proof-translation function $T : \mathbf{P}_{\text{lra}} \to \mathbf{P}$ from such proofs to $\mathcal{L}$ proofs, where applications of rules in $\mathcal{C}$ are translated to applications of corresponding rules for polynomial atoms in $\mathcal{L}$. The liberal translation Lib implements an extension of the operator T, relying on an extension of the invariant below to proofs involving inequalities, as well as logical symbols such as $\Rightarrow$, $\neg$ and $\bot$. For space constraints, we limit our discussion here to proofs involving equality atoms and double implications between such atoms.

Invariant 1 *For all proofs $P \in \mathbf{P}_{\text{lra}}$ the following holds.*

(a) If $P : \Gamma \vdash t = s$, then (i) $T(P) : \Gamma \vdash p = 0$ and (ii) $(p = 0) \equiv (t = s)$.

(b) If $P : \Gamma \vdash t_1 = s_1 \Leftrightarrow t_2 = s_2$, then (i) $T(P) : \Gamma \vdash (c \cdot p_1 - p_2)\!\downarrow = 0$ for some non-zero constant c, (ii) $(p_1 = 0) \equiv (t_1 = s_1)$, and (iii) $(p_2 = 0) \equiv (t_2 = s_2)$.

Thanks to Invariant 1, all translated proofs $T(P)$ are at least as strong as the original proof P (see [15] for a proof).

Proposition 1. *If $P : \Gamma \vdash \varphi$ and $T(P) : \Gamma \vdash p \sim 0$, then $p \sim 0$ entails φ.*

The T operator is defined by structural induction on proofs from $\mathbf{P}_{\text{lra}}$. To give a general idea of this definition, we present here three subcases of T's definition and show how they preserve Invariant 1.

(iff_trans) Let P be a proof of the form:

$$\frac{P_1 : t_1 = s_1 \Leftrightarrow t_2 = s_2 \quad P_2 : t_2 = s_2 \Leftrightarrow t_3 = s_3}{P : t_1 = s_1 \Leftrightarrow t_3 = s_3}\ \textsf{iff\_trans}$$

By assumption of Invariant 1(b) for P_1, we have that $T(P_1) : (c_1 \cdot p_1 - p_2)\!\downarrow = 0$ for some non-zero c_1, where $(p_1 = 0) \equiv (t_1 = s_1)$, and $(p_2 = 0) \equiv (t_2 = s_2)$. Similarly for P_2, we have that $T(P_2) : (c_2 \cdot p_2 - p_3)\!\downarrow = 0$ for some non-zero c_2, where $(p_3 = 0) \equiv (t_3 = s_3)$. The translated proof $T(P)$ is:

$$\frac{\dfrac{T(P_1) : (c_1 \cdot p_1 - p_2)\!\downarrow = 0}{(c_2 \cdot (c_1 \cdot p_1 - p_2))\!\downarrow = 0}\ \textsf{lra\_mult\_c}_{=} \qquad T(P_2) : (c_2 \cdot p_2 - p_3)\!\downarrow = 0}{T(P) : (c_2 \cdot (c_1 \cdot p_1 - p_2) + (c_2 \cdot p_2 - p_3))\!\downarrow = 0}\ \textsf{lra\_add}_{==}$$

Invariant 1(b) holds for P. Note that $(c_2 \cdot (c_1 \cdot p_1 - p_2) + (c_2 \cdot p_2 - p_3))\!\downarrow = ((c_2 \cdot c_1) \cdot p_1 - p_3)\!\downarrow$. This gives us property (i) of Invariant 1(b), noting that $c_2 \cdot c_1$ is non-zero since c_1 and c_2 are both non-zero. Properties (ii) and (iii) hold by assumption.

(iff_symm) Let be P a proof of the form:

$$\frac{P_1 : t_1 = s_1 \Leftrightarrow t_2 = s_2}{P : t_2 = s_2 \Leftrightarrow t_1 = s_1}\ \textsf{iff\_symm}$$

By assumption of Invariant 1(b) for P_1, we have that $T(P_1) : (c \cdot p_1 - p_2)\!\downarrow = 0$ for some non-zero c, where $(p_1 = 0) \equiv (t_1 = s_1)$, and $(p_2 = 0) \equiv (t_2 = s_2)$. The translated proof $T(P)$ is:

$$\frac{T(P_1) : (c \cdot p_1 - p_2)\!\downarrow = 0}{T(P) : (-\frac{1}{c} \cdot (c \cdot p_1 - p_2))\!\downarrow = 0}\ \mathsf{lra\_mult\_c}_{=}$$

To see that Invariant 1(b) holds for P, note that $(-\frac{1}{c} \cdot (c \cdot p_1 - p_2))\!\downarrow = (\frac{1}{c} \cdot p_2 - p_1)\!\downarrow$, giving us property (i). Properties (ii) and (iii) hold by assumption.

(iff_mp) Let be P a proof of the form:

$$\frac{P_1 : t_1 = s_1 \quad P_2 : t_1 = s_1 \Leftrightarrow t_2 = s_2}{P : t_2 = s_2}\ \mathsf{iff\_mp}$$

By assumption of Invariant 1(a) for P_1, we have that $T(P_1) : p_1 = 0$ and $(p_1 = 0) \equiv (t_1 = s_1)$. By assumption of Invariant 1(b) for P_2, we have that $T(P_2) : (c \cdot p_1 - p_2)\!\downarrow = 0$ for some non-zero c, where $(p_2 = 0) \equiv (t_2 = s_2)$. The translated proof $T(P)$ is:

$$\frac{\dfrac{T(P_1) : p_1 = 0}{(c \cdot p_1)\!\downarrow = 0}\ \mathsf{lra\_mult\_c}_{=} \qquad T(P_2) : (c \cdot p_1 - p_2)\!\downarrow = 0}{T(P) : (c \cdot p_1 - (c \cdot p_1 - p_2))\!\downarrow = 0}\ \mathsf{lra\_sub}_{==}$$

It can be shown that Invariant 1(a) holds for P, by noting that $(c \cdot p_1 - (c \cdot p_1 - p_2))\!\downarrow = p_2$. This gives us property (i), and property (ii) holds by assumption.

Example 1 (Proof Translation). Figure 3 shows, in part, a subproof P of a refutation in $\mathcal{C}$ from a set of premises that includes $2x = 2y$ and $y = x + 5$. In the given fragment, which uses rules from Figure 1, one application of iff_mp, followed by one application of eq_trans and another application of iff_mp, yields the contradictory term equation $5 = 0$.

The bottom part of the figure illustrates the correspondence between the conclusion of each shown subproof of P and its associated equivalent polynomial atom. For each proof node, a polynomial inference suffices to prove that atom. As demonstrated in this example, rewrite axioms can be replaced by the trivial identity $0 = 0$, logical modus ponens can be replaced by a subtraction of polynomials (with possible multiplication by a constant), and transitivity for equality can be replaced by polynomial addition.

The translated proof $T(P)$ is given in Figure 4. As a last step in the translation, we cut all portions of the original proof that do not contribute to deriving the conclusion of $T(P)$, such as as P_6 in Figure 3 for instance. Although the example here is simple, P_6 may be in general a highly non-trivial subproof, which when cut, can account for a significant reduction in overall proof size. □

$$\dfrac{\dfrac{\dfrac{P_1 : 2x = 2y \quad \overline{P_2 : 2x = 2y \Leftrightarrow x = y}}{P_3 : x = y} \quad P_4 : y = x + 5}{P_5 : x = x + 5} \quad \dfrac{\vdots}{P_6 : x = x + 5 \Leftrightarrow 5 = 0}}{P : 5 = 0}$$

$$\dfrac{\dfrac{\dfrac{(2x - 2y)\downarrow = 0 \quad \overline{0 = 0}}{(\frac{1}{2} \cdot (2x - 2y) - 0)\downarrow = 0} \quad (y - (x + 5))\downarrow = 0}{(\frac{1}{2} \cdot (2x - 2y) - 0 + y - (x + 5))\downarrow = 0} \quad \dfrac{\vdots}{0 = 0}}{(-1 \cdot (\frac{1}{2} \cdot (2x - 2y) - 0 + y - (x + 5)) - 0)\downarrow = 0}$$

Fig. 3. A proof using only $\mathcal{C}$ rules, as generated by Cvc3, followed by corresponding polynomial inferences.

$$\dfrac{\dfrac{T(P_1) : 2x - 2y = 0}{T(P_3) : x - y = 0} \quad T(P_4) : -x + y - 5 = 0}{T(P) : 5 = 0}$$

Fig. 4. Translation of the proof P from Figure 3.

4 Experimental Results

To evaluate the various translations experimentally, we looked at all the QF_LRA and QF_RDL unsatisfiable benchmarks from SMT-LIB Version 1.2.[5] Our results contain no comparisons with other proof checkers besides LFSC here, although previous work [14] has shown that for other logics, LFSC is highly competitive with other SMT proof checkers such as Fx7 [11]. A potential candidate for this work was a former system developed by Ge and Barrett that used the HOL Light prover as a proof checker for Cvc3 [7]. Unfortunately, that system, which was never tested on QF_LRA benchmarks and was not kept in sync with the latest developments of Cvc3, currently breaks on most of these benchmarks. While we expect that it could be fixed, the required amount of effort is beyond the scope of this work.

We ran our experiments on a Linux machine with two 2.67GHz 4-core Xeon processors and 8GB of RAM. We will discuss benchmarks for which Cvc3 could generate a proof within a timeout of 900 seconds, that is, 161 of the 317 unsatisfiable QF_LRA benchmarks, and 40 of the 113 unsatisfiable QF_RDL benchmarks.

We collected runtimes for the following four main configurations of Cvc3.

cvc:	Default, solving benchmarks but with no proof generation.
cvcpf:	Solving with proof generation in Cvc3's native format.
lit:	Solving with proof generation and literal translation to LFSC.
lib:	Solving with proof generation and liberal translation to LFSC.

We also ran a fifth configuration, **litNT**, for the purpose of isolating the non-theory component of proof sizes and checking times. This configuration trusts all theory lem-

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

Benchmark		**Solve + (Pf Gen) + (Pf Conv) (sec)**					**Proof Size (MB)**				**Pf Check Time (sec)**			
Class	#	cvc	cvcpf	lit	lib	litNT	cvcpf	lit	lib	litNT	lit	lib	litNT	T%
check-lra	1	0.1	0.2	0.3	0.2	0.2	0.3	0.5	0.1	0.1	0.1	0.0	0.03	79%
check-rdl	1	0.1	0.1	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	48%
clock_synch	18	11.7	21.0	21.8	21.7	21.5	9.1	14.8	13.0	12.7	2.6	2.3	2.3	17%
gasburner	19	4.0	7.5	8.6	7.8	6.8	8.5	13.9	7.7	6.8	2.5	1.6	1.2	46%
pursuit	8	16.6	26.6	26.3	26.3	25.7	3.9	5.1	3.6	3.4	0.8	0.7	0.6	36%
sal	31	1584.8	3130.5	3254.3	3239.8	3285.9	718.6	537.1	472.2	452.1	275.6	269.0	262.0	6%
scheduling	8	281.8	322.0	322.9	322.1	321.1	18.4	25.1	17.8	18.3	3.7	2.9	2.6	37%
spider	35	10.2	16.7	17.4	17.4	16.8	10.1	12.7	11.1	10.8	2.3	2.4	2.0	15%
tgc	21	31.5	54.8	55.9	55.4	54.3	21.0	22.7	16.9	16.6	4.2	3.4	3.1	16%
TM	1	17.6	29.4	29.3	29.0	29.1	1.3	2.7	2.7	2.7	0.4	0.4	0.4	0%
tta_startup	25	29.7	65.9	68.4	68.5	67.9	38.8	43.6	43.2	42.9	5.4	5.6	5.3	3%
uart	9	1074.2	1387.4	1391.2	1434.7	1379.1	118.9	102.4	76.6	72.2	42.7	37.0	34.5	13%
windowreal	24	20.6	40.6	41.6	41.7	41.4	20.7	22.1	21.9	21.7	2.8	2.9	2.9	3%
Total	201	3082.9	5102.6	5238.0	5264.6	5249.7	969.4	802.8	686.9	660.2	343.2	328.1	316.8	8%

Table 1. Cumulative results, grouped by benchmark class. Column 2 gives the numbers of benchmarks in each class. Columns 3 through 8 give Cvc3's (aggregate) runtime for each of the five configurations. Columns 9 through 13 show the proof sizes for each of the 5 proof-producing configurations. Columns 14 through 17 show LFSC proof checking times. The last column gives the percentage of proof nodes found beneath theory lemmas in Cvc3's native proofs.

mas treating them like premises, but otherwise behaves like **lit** (and so also like Lib which differs from Lit only on theory lemmas). Comparisons with **litNT** are useful because the liberal translation works solely by compacting the theory-specific portion of a proof. Hence, their effectiveness is expected to be correlated with the amount of *theory content* of a proof. We measure that as the percentage of nodes in a Cvc3 proof belonging to the (sub)proof of a theory lemma. For this data set, the average theory content is very low, about 8.3%.

Table 1 shows a summary of our results for various classes of benchmarks.[6] As can be seen there, Cvc3's solving times are on average 1.65 times faster than solving with native proof generation. The translation to LFSC proofs adds additional overhead, which is however less than 3% on average for all translations.

The scatter plots in Figure 5 are helpful in comparing proof sizes for the various configurations.[7] The first plot compares proofs in Cvc3 native format against their literal translation Lit. Notice that, except for a couple of outliers, Lit suffers only a small constant overhead which we believe is due to structural differences between the Cvc3 and the LFSC proof languages.

The second plot shows that the liberal translation Lib introduces constant compression factors over the literal translation. A number of benchmarks in our test set do not benefit from the Lib translation. Such benchmarks are not heavily dependent on theory reasoning, having a theory content of less than 2%. In contrast, for benchmarks with higher theory content, Lib is effective at proof compression. Over the set of all benchmarks with *enough* theory content, quantified as 10% or more, Lib compresses

[6] Detailed results are available at `http://clc.cs.uiowa.edu/CPP11`.

[7] These plots show only the data for proof sizes less than or equal to 5MB. The general trends shown by these plots are preserved with the addition of larger benchmarks.

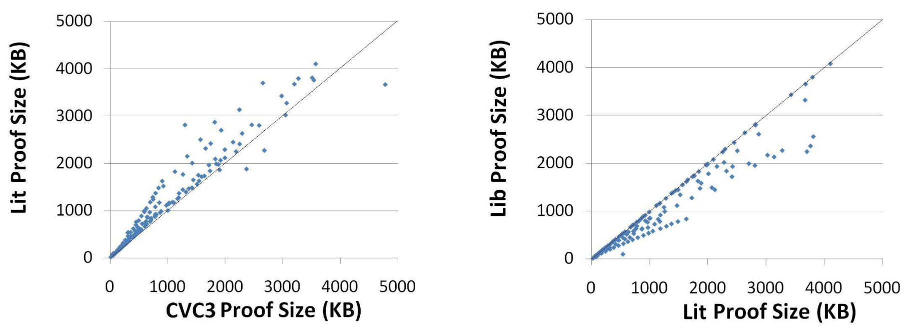

Fig. 5. Comparing proof sizes.

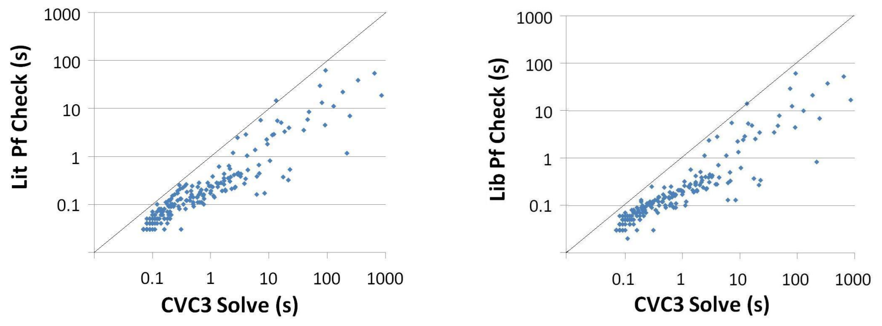

Fig. 6. Solving times vs. proof checking times. Values are on log-log scale.

proof sizes an average of 24%—i.e, a Lib proof on average uses 24% less space than its corresponding Lit proof. When focusing on theory lemma subproofs, by subtracting proofs sizes in **litNT** from both **lit** and **lib**, the average compression goes up significantly, to 81.3%; that is to say, after compaction, theory lemma subproofs are a factor of 5.3 smaller than in the original proof.

Interestingly, in all plots 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.

We compared the proof checking times of Lit vs. Lib, using the LFSC checker. Perhaps unsurprisingly, the scatter plot (not shown here) is very similar to the corresponding one in Figure 5. Over benchmarks with enough theory content, checking Lib proofs is on average 1.14 times faster than checking the corresponding Lit proofs. Looking just at proofs of theory lemmas, this time by subtracting the checking times of **litNT**, reveals that proof checking times are 2.33 times faster for Lib than for Lit.

It is generally expected that proof checking should be substantially faster than proof generation or even just solving. This is generally the case for both Lit and Lib when proof checking using compiled side conditions. Compared against Cvc3's solving times alone, LFSC proof checking times are 8.98 times faster with Lit proofs, and 9.4 times faster with Lib proofs. A more detailed comparison (given on a logarithmic scale) can be seen in Figure 6.

5 Conclusion and Further Work

We have investigated alternative proof systems for quantifier-free Linear Real Arithmetic in the LFSC framework. Proofs in these systems were produced by translating in LFSC proofs generated by the Cvc3 SMT solver. The flexibility of LFSC enabled us to consider, without modification to LFSC itself, two translations differing in their degree of faithfulness to native Cvc3 proofs and compaction of arithmetic reasoning steps. We have demonstrated that the second of these translations is effective in compressing trace-style proofs for LRA into proofs given by a set of coefficients and computational inferences, leading to faster proof checking times. In addition, our experiments demonstrate that our approach to proof checking scales well. For benchmarks taking more than 30 seconds to solve, Lit proof checking times were approximately 11 times faster than solving times on average.

We plan to integrate native LFSC proof generation ability into CVC3's successor, CVC4, with support for other logics beyond QF_LRA. For LRA, this will include use of the $\mathcal{L}$ calculus as the LFSC signature for checking proofs produced by CVC4's simplex procedure. The use of a simplex procedure will allow coefficients for certain proofs fragments to be extracted directly from the solver. We are also investigating other applications of the LFSC meta-framework, including using the LFSC checker to generate Craig interpolants in selected theories by type inference over proofs in these theories.

We also plan to encourage the SMT community to consider adopting LFSC as a common format, by making available a public release of the LFSC toolset, which will provide an intuitive high-level interface for specifying proof systems that hides the concrete syntax of LFSC, as well as an API for generating proofs.

References

1. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.smt-lib.org`, 2010.
2. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.
3. L. Bauer, S. Garriss, J. McCune, M. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, Sept. 2005.
4. S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.
5. J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In A. Aiken, editor, *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 412–423, 2010.

6. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. *ACM Transactions on Computational Logic*, 2011. to appear.
7. Y. Ge and C. Barrett. Proof translation and SMT-LIB benchmark certification: A preliminary report. In *Proceedings of SMT'08*, 2008.
8. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.
9. D. Lee, K. Crary, and R. Harper. Towards a Mechanized Metatheory of Standard ML. In *Proc. 34th ACM Symposium on Principles of Programming Languages*, pages 173–184. ACM Press, 2007.
10. G. Li and G. Gopalakrishnan. Scalable SMT-Based Verification of GPU Kernel Functions. In A. van der Hoek, editor, *FSE '10: Proceedings of the 2010 ACM SIGSOFT conference on Foundations of software engineering*, 2010. to appear.
11. M. Moskal. Rocket-fast proof checking for smt solvers. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 486–500, Berlin, Heidelberg, 2008. Springer-Verlag.
12. G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.
13. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. 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, Nov. 2006.
14. D. Oe, A. Reynolds, and A. Stump. Fast and flexible proof checking for SMT. In *Proceedings of SMT'09*, 2009.
15. A. Reynolds, L. Hadarean, C. Tinelli, Y. Ge, A. Stump, and C. Barrett. CVC3 Proof Conversion to LFSC, 2010. Companion report, available from `http://clc.cs.uiowa.edu/CPP11/`.
16. W. Steiner and B. Dutertre. SMT-Based Formal Verification of a *TEthernet* Synchronization Function. In S. Kowalewski and M. Roveri, editors, *Formal Methods for Industrial Critical Systems - 15th International Workshop, FMICS 2010*, pages 148–163, 2010.
17. A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

A $\mathcal{C}$ Proof Rules

The following is a representative list of rules in the $\mathcal{C}$ calculus. The letters c and t, possibly with subscripts, denote rational constants and arithmetic terms, respectively.

A.1 Core Rules

$$\frac{v \vee \varphi_1 \quad \neg v \vee \varphi_2}{\varphi_1 \vee \varphi_2}\ \textsf{bool\_res} \qquad \frac{\begin{array}{c}[\varphi_1 \wedge \ldots \wedge \varphi_n] \\ \vdots \\ \bot\end{array}}{\neg\varphi_1 \vee \ldots \neg \vee \varphi_n}\ \textsf{learned\_clause}$$

A.2 Rewrite Axioms

$$\frac{\{0 \nsim c\}}{(0 \sim c) \Leftrightarrow \bot}\ \textsf{const\_pred}_1 \qquad \frac{\{0 \sim c\}}{(0 \sim c) \Leftrightarrow \top}\ \textsf{const\_pred}_2$$

$$\frac{\{c \text{ positive}\}}{t_1 < t_2 \Leftrightarrow c \cdot t_1 < c \cdot t_2}\ \textsf{mult\_ineqn} \qquad \frac{\{c \text{ non-zero}\}}{t_1 = t_2 \Leftrightarrow c \cdot t_1 = c \cdot t_2}\ \textsf{mult\_eqn}$$

$$\frac{}{t_1 \sim t_2 \Leftrightarrow 0 \sim t_2 - t_1}\ \textsf{right\_minus\_left} \qquad \frac{}{t_1 \sim t_2 \Leftrightarrow t_1 + t_3 \sim t_2 + t_3}\ \textsf{plus\_pred}$$

$$\frac{}{t_1 > t_2 \Leftrightarrow t_2 < t_1}\ \textsf{flip\_ineq} \qquad \frac{}{\neg(t_1 < t_2) \Leftrightarrow t_1 \geq t_2}\ \textsf{negated\_ineq}$$

$$\frac{\{c_1 < c_2\}}{0 < c_1 + t \Rightarrow 0 < c_2 + t}\ \textsf{weaker\_ineq}$$

A.3 Propositional Rules

$$\frac{\varphi_1 \Leftrightarrow \varphi_2 \quad \varphi_2 \Leftrightarrow \varphi_3}{\varphi_1 \Leftrightarrow \varphi_3}\ \textsf{iff\_trans} \qquad \frac{\varphi_1 \quad \varphi_1 \Leftrightarrow \varphi_2}{\varphi_2}\ \textsf{iff\_mp}$$

$$\frac{\varphi_1 \Leftrightarrow \varphi_2}{\varphi_2 \Leftrightarrow \varphi_1}\ \textsf{iff\_symm} \qquad \frac{}{\varphi \Leftrightarrow \varphi}\ \textsf{iff\_refl}$$

$$\frac{\varphi_1 \Rightarrow \varphi_2 \quad \varphi_2 \Rightarrow \varphi_3}{\varphi_1 \Rightarrow \varphi_3}\ \textsf{impl\_trans} \qquad \frac{\varphi_1 \quad \varphi_1 \Rightarrow \varphi_2}{\varphi_2}\ \textsf{impl\_mp}$$

A.4 Equality Rules

$$\frac{}{t_1 = t_1}\ \textsf{refl}$$

$$\frac{t_1 = t_2 \quad t_3 = t_4}{t_1 \sim t_3 \Leftrightarrow t_2 \sim t_4}\ \textsf{congr\_1} \qquad \frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \textsf{eq\_trans}$$

$$\frac{t_1 = t_2 \quad t_3 = t_4}{t_1 \bowtie t_3 = t_2 \bowtie t_4}\ \textsf{congr\_2} \qquad \frac{t_1 = t_2}{t_2 = t_1}\ \textsf{eq\_symm}$$

A.5 Theory Rules

$$\frac{t_1 < t_2 \quad t_2 < t_3}{t_1 < \ t_3}\ \textsf{gt\_trans} \qquad \frac{t_1 \geq t_2 \quad t_1 \leq t_2}{t_1 = t_2}\ \textsf{gt\_antisym}$$

$$\frac{t_1 < t_2 \quad t_3 < t_4}{t_1 + t_3 < \ t_2 + t_4}\ \textsf{add\_inequalities} \qquad \frac{}{-t = (-1) \cdot t}\ \textsf{uminus\_to\_mult}$$

$$\frac{}{(t_1 - t_2) = t_1 + (-1 \cdot t_2)}\ \textsf{minus\_to\_plus} \qquad \frac{\{t' \text{ canonical form of } t\}}{t = t'}\ \textsf{canon}$$

A.6 CNF Conversion Rules

$$\frac{}{(\varphi_1 \Rightarrow \varphi_2) \vee \varphi_1}\ \textsf{CNF\_imp\_0} \qquad \frac{}{(\varphi_1 \Rightarrow \varphi_2) \vee \neg\varphi_2}\ \textsf{CNF\_imp\_1}$$

$$\frac{}{\neg(\varphi_1 \Rightarrow \varphi_2) \vee \neg\varphi_1 \vee \varphi_2}\ \textsf{CNF\_imp\_2}$$

$$\frac{}{(\varphi_1 \Leftrightarrow \varphi_2) \vee \varphi_1 \vee \varphi_2}\ \textsf{CNF\_iff\_0} \qquad \frac{}{(\varphi_1 \Leftrightarrow \varphi_2) \vee \neg\varphi_1 \vee \neg\varphi_2}\ \textsf{CNF\_iff\_1}$$

$$\frac{}{\neg(\varphi_1 \Leftrightarrow \varphi_2) \vee \neg\varphi_1 \vee \varphi_2}\ \textsf{CNF\_iff\_2} \qquad \frac{}{\neg(\varphi_1 \Leftrightarrow \varphi_2) \vee \varphi_1 \vee \varphi_2}\ \textsf{CNF\_iff\_3}$$

$$\frac{}{\neg(\varphi_1 \wedge \ldots \wedge \varphi_n) \vee \varphi_i}\ \textsf{CNF\_and\_mid} \qquad \frac{}{(\varphi_1 \wedge \ldots \wedge \varphi_n) \vee \neg\varphi_1 \vee \ldots \vee \neg\varphi_n}\ \textsf{CNF\_and\_final}$$

$$\frac{}{(\varphi_1 \vee \ldots \vee \varphi_n) \vee \neg\varphi_i}\ \textsf{CNF\_or\_mid} \qquad \frac{}{\neg(\varphi_1 \vee \ldots \vee \varphi_n) \vee \varphi_1 \vee \ldots \vee \varphi_n}\ \textsf{CNF\_or\_final}$$

$$\frac{}{\neg ite(\phi, \varphi_1, \varphi_2) \vee \phi \vee \varphi_2}\ \textsf{CNFITE\_0} \qquad \frac{}{ite(\phi, \varphi_1, \varphi_2) \vee \phi \vee \neg\varphi_2}\ \textsf{CNFITE\_1}$$

$$\frac{}{ite(\phi, \varphi_1, \varphi_2) \vee \neg\phi \vee \neg\varphi_1}\ \textsf{CNFITE\_2} \qquad \frac{}{\neg ite(\phi, \varphi_1, \varphi_2) \vee \neg\phi \vee \varphi_1}\ \textsf{CNFITE\_3}$$

$$\frac{}{ite(\phi, \varphi_1, \varphi_2) \vee \phi \vee \neg\varphi_1 \vee \neg\varphi_2}\ \textsf{CNFITE\_4} \qquad \frac{}{\neg ite(\phi, \varphi_1, \varphi_2) \vee \varphi_1 \vee \varphi_2}\ \textsf{CNFITE\_5}$$

B $\mathcal{L}$ Specific Proof Rules

In the proof rules below, the expression $p{\downarrow}$ denotes the result of normalizing the polynomial expression p. The normalization is done by the rules side condition, which is however left implicit here to keep the notation uncluttered.

B.1 Axioms

$$\frac{}{0=0}\ \mathsf{lra\_axiom}_{=} \qquad \frac{\{c>0\}}{c>0}\ \mathsf{lra\_axiom}_{>}$$

$$\frac{\{c\geq 0\}}{c\geq 0}\ \mathsf{lra\_axiom}_{\geq} \qquad \frac{\{c\neq 0\}}{c\neq 0}\ \mathsf{lra\_axiom}_{\neq}$$

B.2 Equality Deduction Rule

$$\frac{p\geq 0 \quad p'\geq 0 \quad \{p+p'=0\}}{p=0}\ \mathsf{lra}{\geq}{\geq}\mathsf{to}{=}$$

B.3 Contradiction Rules

$$\frac{p=0 \quad \{p\neq 0\}}{\bot}\ \mathsf{lra\_contra}_{=} \qquad \frac{p>0 \quad \{p\not>0\}}{\bot}\ \mathsf{lra\_contra}_{>}$$

$$\frac{p\geq 0 \quad \{p<0\}}{\bot}\ \mathsf{lra\_contra}_{\geq} \qquad \frac{p\neq 0 \quad \{p=0\}}{\bot}\ \mathsf{lra\_contra}_{\neq}$$

B.4 Multiplication Rules

$$\frac{p=0}{(c\cdot p)\!\downarrow\,=0}\ \mathsf{lra\_mult\_}c_{=} \qquad \frac{p>0 \quad \{c>0\}}{(c\cdot p)\!\downarrow\,>0}\ \mathsf{lra\_mult\_}c_{>}$$

$$\frac{p\geq 0 \quad \{c\geq 0\}}{(c\cdot p)\!\downarrow\,\geq 0}\ \mathsf{lra\_mult\_}c_{\geq} \qquad \frac{p\neq 0 \quad \{c\neq 0\}}{(c\cdot p)\!\downarrow\,\neq 0}\ \mathsf{lra\_mult\_}c_{\neq}$$

B.5 Addition Rules

$$\frac{p_1=0 \quad p_2=0}{(p_1+p_2)\!\downarrow\,=0}\ \mathsf{lra\_add}_{==} \qquad \frac{p_1>0 \quad p_2>0}{(p_1+p_2)\!\downarrow\,>0}\ \mathsf{lra\_add}_{>>}$$

$$\frac{p_1\geq 0 \quad p_2\geq 0}{(p_1+p_2)\!\downarrow\,\geq 0}\ \mathsf{lra\_add}_{\geq\geq} \qquad \frac{p_1=0 \quad p_2>0}{(p_1+p_2)\!\downarrow\,>0}\ \mathsf{lra\_add}_{=>}$$

$$\frac{p_1=0 \quad p_2\geq 0}{(p_1+p_2)\!\downarrow\,\geq 0}\ \mathsf{lra\_add}_{=\geq} \qquad \frac{p_1>0 \quad p_2\geq 0}{(p_1+p_2)\!\downarrow\,>0}\ \mathsf{lra\_add}_{>\geq}$$

$$\frac{p_1=0 \quad p_2\neq 0}{(p_1+p_2)\!\downarrow\,\neq 0}\ \mathsf{lra\_add}_{=\neq}$$

B.6 Subtraction Rules

$$\frac{p_1 = 0 \quad p_2 = 0}{(p_1 - p_2)\!\downarrow\, = 0}\ \textsf{lra\_sub}_{==} \qquad \frac{p_1 > 0 \quad p_2 = 0}{(p_1 - p_2)\!\downarrow\, > 0}\ \textsf{lra\_sub}_{>=}$$

$$\frac{p_1 \geq 0 \quad p_2 = 0}{(p_1 - p_2)\!\downarrow\, \geq 0}\ \textsf{lra\_sub}_{\geq=} \qquad \frac{p_1 \neq 0 \quad p_2 = 0}{(p_1 - p_2)\!\downarrow\, \neq 0}\ \textsf{lra\_sub}_{\neq=}$$

B.7 Term Normalization Rules

In the rules below c_{t} and c_{p} denote the same rational constant, in one case considered of term type and in the other as of polynomial type (similarly for the variables v_{t} and v_{p}).

$$\frac{}{c_{\mathrm{t}} = c_{\mathrm{p}}}\ \textsf{poly\_norm\_const} \qquad \frac{t_1 = p_1 \quad t_2 = p_2}{t_1 + t_2 = (p_1 + p_2)\!\downarrow}\ \textsf{poly\_norm}_{+}$$

$$\frac{}{v_{\mathrm{t}} = v_{\mathrm{p}}}\ \textsf{poly\_norm\_var} \qquad \frac{t_1 = p_1 \quad t_2 = p_2}{t_1 - t_2 = (p_1 - p_2)\!\downarrow}\ \textsf{poly\_norm}_{-}$$

$$\frac{t = p}{c_{\mathrm{t}} \cdot t = (c_{\mathrm{p}} \cdot p)\!\downarrow}\ \textsf{poly\_norm}_{c\cdot} \qquad \frac{t = p}{t \cdot c_{\mathrm{t}} = (p \cdot c_{\mathrm{p}})\!\downarrow}\ \textsf{poly\_norm}_{\cdot c}$$

B.8 Equation Normalization Rules

$$\frac{t_1 = t_2 \quad t_1 - t_2 = p}{p = 0}\ \textsf{poly\_norm}_{=} \qquad \frac{t_1 \neq t_2 \quad t_2 - t_1 = p}{p \neq 0}\ \textsf{poly\_norm}_{\neq}$$

$$\frac{t_1 > t_2 \quad t_1 - t_2 = p}{p > 0}\ \textsf{poly\_norm}_{>} \qquad \frac{t_1 < t_2 \quad t_2 - t_1 = p}{p > 0}\ \textsf{poly\_norm}_{<}$$

$$\frac{t_1 \geq t_2 \quad t_1 - t_2 = p}{p \geq 0}\ \textsf{poly\_norm}_{\geq} \qquad \frac{t_1 \leq t_2 \quad t_2 - t_1 = p}{p \geq 0}\ \textsf{poly\_norm}_{\leq}$$