Reflected Decision Procedures in Lean

Seul Baek

January 29, 2019

Submitted in partial fulfillment of the requirements for the degree of
M.S. in Logic, Computation and Methodology
Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA, USA

Acknowledgements

I would like to thank my advisor Jeremy Avigad, not only for his astute guidance and the patience with which he supervised my development of ideas, but also for his enthusiastic encouragements; whenever I am down and full of skepticism, it's an afternoon talk with Jeremy that makes me excited again for new challenges. I'd like to thank my committee members Tom Hales and André Platzer for their insightful comments, especially for the pointers to future work that I look forward to explore. I owe thanks to Mario Carneiro, discussions with whom provided many key ideas for this thesis (and are always entertaining). I'd like to thank the members of the Lean developer community, whose friendly help and expertise I could always count on. I am grateful for my family, without whose love and support I could not have come this far.

Contents

1 Introduction **4**

2 Preliminaries **4**

2.1 Quantifier Elimination 4

2.2 Cooper's Algorithm 8

2.3 Computational Reflection 16

3 Implementation **20**

3.1 Planning 20

3.2 Reification 26

3.3 Discharging Goals 31

4 Test Results **34**

5 Conclusion and Future Work **37**

6 Related Works **38**

7 References **38**

A List of Test Cases **39**

1 Introduction

The term 'computational reflection' can mean very different things depending on the context. In the broadest sense, it is defined as "the activity performed by a computational system when reasoning about (and by that possibly affecting) itself" [7]. In this thesis, we use the term more narrowly to refer to the method of efficiently rewriting terms in interactive theorem provers via computation, following Boyer and Moore [3].

This thesis aims to answer two main questions: (1) What are the main steps and challenges in implementing reflected decision procedures in Lean? (2) Can reflected decision procedures in Lean solve practical problems? Although the basic principle of computational reflection is well-understood, it is difficult to predict its effectiveness for a new theorem prover because the technique is highly dependent on prover-specific features, such as its metalanguage and kernel evaluation. This is particularly true for Lean, which is unusual in having its object language double as its metalanguage, and whose existing examples of computational reflection are still small and experimental. To answer these questions, this thesis presents `lia`, a new Lean tactic which implements Cooper's algorithm, and uses it as an example to explore the challenges and feasibility of reflected decision procedures in Lean.

The following sections are organized as follows: Section 2 covers the basic background for quantifier elimination, Cooper's algorithm, and computational reflection. Section 3 shows how reflected decision procedures can be implemented in Lean by tactic programming. Section 4 provides empirical data from performance tests and evaluates them against initial expectations. Section 5 sums up the results and discusses potential extensions of the work.

2 Preliminaries

2.1 Quantifier Elimination

This section presents a brief overview of quantifier elimination, a technique pioneered by Tarski [10] which forms the basis of various decision procedures. The core insight of quantifier elimination, in rough outlines, is as follows: suppose you have a program which, given any formula, returns a quantifier-free formula equivalent to the input. If you run this program with a sentence as input, you get a ground propositional formula as output (or something that can be turned into a ground propositional formula, in a sense that will be made precise shortly), whose truth value is often decidable. Since the input and output have the same truth values, this program can be used to decide the truth value of a sentence.

In the following, we discuss the precise circumstances under which quantifier elimination can decide the truth value of sentences. Before getting into details, we first fix a few terminologies:

Definition 2.1. A structure S is **decidable** if there is an effective procedure that decides $S \vDash \phi$ for any sentence ϕ in the language of S.

Definition 2.2. For any structure S and formulas ϕ, ψ in its language, if ϕ and ψ are S-equivalent (i.e., $S \vDash \phi \leftrightarrow \psi$) and ψ is quantifier-free, then ψ is a **quantifier-free equivalent**, or **QFE**, of ϕ.

Note that whether a formula ψ is a QFE of another formula ϕ depends on the structure S in which they are evaluated. This choice of background structure will always be clear from context whenever we discuss QFEs.

Definition 2.3. For any structure S, an effective procedure which computes a QFE of any formula ϕ in the language of S is a **quantifier-elimination procedure**, or **QEP**, of S.

Not all structures that have QEPs are decidable. In order to decide a structure S with its QEP, S has to satisfy a number of additional requirements:

1. For any ground atom a of S, $S \vDash a$ is decidable.

2. The signature of S includes at least one constant.

Although not necessary, it is convenient to assume that the structure also allows elimination of negated atoms. This property holds for many useful structures and significantly simplifies proofs:

3. For any atom a of S, we can compute a formula ϕ that is negation-free, quantifier-free, and equivalent to $\neg a$.

For the rest of the section, we fix S to an arbitrary structure that satisfies conditions 1-3, and L to the language of S.

Theorem 2.1. *If S has a QEP, S is decidable.*

Proof. Let ϕ be an arbitrary sentence of L. We show that $S \vDash \phi$ is decidable. Let ψ be the QFE of ϕ computed by the QEP, and ψ' the formula obtained by replacing all free variables of ψ with a constant c (which exists by assumption). ψ' has neither quantifiers nor free variables, which means it is a propositional combination of ground atoms. Since $S \vDash a$ is decidable for any atom a in ψ' by assumption, $S \vDash \psi'$ is decidable. Therefore, it suffices to show that ψ' is S-equivalent to ϕ, which holds by

$$\begin{aligned} & S \vDash \psi' \\ \iff & S, v \vDash \psi \\ \iff & S \vDash \psi \\ \iff & S \vDash \phi \end{aligned}$$

where v is the valuation that maps every free variable of ψ to the value of c in S. The second equivalence holds because the truth value of ψ is constant under all valuations: since ϕ is a sentence whose truth value is unaffected by valuations, the same holds for ψ, which is S-equivalent to ϕ. $\square$

For the remainder of the section, we show that the existence of a QEP for S can be derived from successively weaker premises. For concision, we present constructive proofs of existence in lieu of explicitly describing effective procedures. In other words, the two lemmas

- There exists an effective procedure which computes an x such that Δ, given Γ.
- If Γ holds, there exists an x such that Δ.

are equivalent, provided that the proof of the latter is constructive. In particular, the two lemmas

- There exists a QEP of S.
- For any $\phi \in L$, there exists a QFE of ϕ.

are equivalent, and we show the various conditions under which the latter can be proven constructively.

Theorem 2.2. *Suppose that for any variable x and quantifier-free $\phi \in L$, there exists a QFE of $\exists x.\phi$. Then for any $\phi \in L$, there exists a QFE of ϕ.*

Proof. By induction on ϕ. The only case not immediately obtained by replacing subformulas of ϕ with their respective QFEs (which exist by IH) is $\phi = \exists x.\chi$.[1] Let ψ be a QFE of χ (which also exists by IH). Since ψ is quantifier-free, there exists a QFE of $\exists x.\psi$ by assumption, which is also a QFE of $\exists x.\chi$ by equivalence of ψ and χ. $\square$

Theorem 2.2 implies that any procedure which eliminates a single existential quantifier can be 'lifted' to a general QEP. This property is highly useful, since proving the decidability of a structure boils down to the relatively simple task of showing the steps for removing a single quantifier. We can further facilitate proofs of decidability by assuming that the bodies of existentially quantified formulas satisfy various additional properties.

Definition 2.4. A formula is a **literal** if it is either an atom or a negated atom. A formula is in **negation normal form**, or **NNF**, if it consists entirely of $\bot, \top, \wedge, \vee$, and literals.

Theorem 2.3. *For any quantifier-free $\phi \in L$, there exists a a formula of L that is in NNF and equivalent to ϕ.*

Proof. By induction on the size of ϕ, measured by the total number of logical constants and atomic formulas in a formula. Excluding the cases that are either itself in NNF or can be immediately transformed into NNF by replacing each subformula with its NNF-equivalent, it remains to show the conclusion for cases where ϕ is $\neg\neg\psi$, $\neg(\psi \vee \chi)$, or $\neg(\psi \wedge \chi)$. For each of the cases, ψ_{nnf}, $(\neg\psi)_{nnf} \wedge (\neg\chi)_{nnf}$, and $(\neg\psi)_{nnf} \vee (\neg\chi)_{nnf}$ are the respective NNF-equivalents, where ϕ_{nnf} denotes the NNF-equivalent of ϕ whose existence is implied by IH. Note that the application of IH in the latter cases is justified since the size of $\neg(\psi * \chi)$ is greater than that of either $\neg\psi$ or $\neg\psi$ by at least 2, where $*$ is either $\vee$ or $\wedge$. $\square$

Theorem 2.4. *For any quantifier-free $\phi \in L$, there exists a $\psi \in L$ such that ψ is negation-free, quantifier-free, and equivalent to ϕ.*

Proof. By Theorem 2.3, there exists a $\phi_{nnf} \in L$ which is in NNF and equivalent to ϕ. By definition of NNF, the only negations in ϕ_{nnf} are negated atoms. The result of replacing every negated atom $\neg a$ in ϕ_{nnf} with the equivalent negation-free and quantifier-free formula (which exists by assumption on S) has all the requisite properties. $\square$

[1]For simplicity, we assume that the set of logical constants of L is $\{\top, \bot, \neg, \vee, \wedge, \exists\}$.

Theorem 2.5. *Suppose that for any variable x and $\phi \in L$ such that ϕ is negation-free and quantifier-free, there exists a QFE of $\exists x.\phi$. Then for any variable x and quantifier-free $\phi \in L$, there exists a QFE of $\exists x.\phi$.*

Proof. By Theorem 2.4, there exists a $\psi \in L$ such that ψ is negation-free, quantifier-free, and equivalent to ϕ. By assumption, there exists a QFE of $\exists x.\psi$, which is also a QFE of $\exists x.\phi$. □

Corollary 2.5.1. *Suppose that for any variable x and $\phi \in L$ such that ϕ is negation-free and quantifier-free, there exists a QFE of $\exists x$. Then S is decidable.*

Proof. Follows from Theorems 2.1, 2.2, and 2.5. □

2.2 Cooper's Algorithm

The structure of linear integer arithmetic is a structure of the integers whose terms and atoms are defined inductively as:

$$t := \ i \ \Big| \ i * x \ \Big| \ t + t$$

$$a := t \leq t \ \Big| \ i \mid t \ \Big| \ i \nmid t$$

In the definition above and the rest of the section, t, s range over terms, i, j, k over integers constants, x, y, z over integer variables, and a over atoms, all of them with or without subscripts (other symbols may also be used when they are informative – e.g., l for integers that serve as lower bounds). '$i \nmid t$' denotes 'i does not divide t', and all symbols are given the usual interpretations for integers. Examples of true sentences of the structure include $\exists x.\ 5 + x \leq 11$ and $\forall x.\ 3 \mid 6 * x$. We henceforth call this structure S_{LIA}, and its language L_{LIA}.

There is a significant practical incentive for deciding S_{LIA}, because sentences in L_{LIA} are frequently encountered in theorem proving involving integers, especially in software verification. The structure was first shown to be decidable by Presburger [9] in 1929, after whom it is also known as Presburger arithmetic. In 1972, Cooper [5] introduced a more efficient decision procedure which avoids expensive DNF transformations, which is the algorithm that `lia` is based on and we will review in this section. The proof mostly follows that of Harrison [6], but we use syntaxes similar to that of Nipkow [8] to stay closer to definitions used in the Lean implementation.

By Corollary 2.5.1 and the background assumptions of Section 2.1, a structure is decidable if

1. Its ground atoms are decidable.

2. It has at least one constant symbol.

3. Its negated atoms can be eliminated.

4. The following can be proven constructively: for any variable x and formula ϕ in its language such that ϕ is negation-free and quantifier-free, there exists a QFE of ϕ.

Conditions 1 and 2 clearly hold for S_{LIA}, and the equivalences

$$\begin{gathered}\neg x \leq y \leftrightarrow y + 1 \leq x \\ \neg x \mid y \leftrightarrow x \nmid y \\ \neg x \nmid y \leftrightarrow x \mid y\end{gathered}$$

suffice to show condition 3. For the rest of the section, we prove condition 4 for S_{LIA}, where the construction of QFE follows Cooper's algorithm. For brevity, we implicitly assume S_{LIA} and L_{LIA} for the remainder of the section. For instance, by 'any formula,' we mean any formula of L_{LIA}; when formulas ϕ and ψ are 'equivalent,' they are S_{LIA}-equivalent (i.e., $S_{LIA} \vDash \phi \leftrightarrow \psi$).

The high-level proof sketch for the section is as follows: First, we define a notion of normalcy for formulas, and show that for any variable x and formula ϕ such that ϕ is free of negations and quantifiers, there exists a normal formula ψ such that $\exists x.\phi$ and $\exists x.\psi$ are equivalent. Second, we prove that for any normal formula ψ, there exists a QFE of $\exists x.\psi$. Together, this amounts to showing the existence of a QFE of $\exists x.\phi$ for arbitrary variable x and formula ϕ whenever ϕ is free of negations and quantifiers. We begin with defining a suitable normal form:

Definition 2.5. For any variable x and formula ϕ, ϕ is ***x*-normal** if it is negation-free, quantifier-free, and each atom in ϕ is in one of the following forms:

$$\begin{array}{l} t \leq 1 * x \\ t \leq 0 \\ t \leq -1 * x \\ i \mid 1 * x + t \\ i \mid t \\ i \nmid 1 * x + t \\ i \nmid t \end{array}$$

where t has no occurrences of x.

Theorem 2.6. *For any variable x and formula ϕ such that ϕ is negation-free and quantifier-free, there exists an x-normal formula ψ such that $\exists x.\phi$ and $\exists x.\psi$ are equivalent.*

Proof. By grouping and factoring terms, we can rewrite each atom of ϕ into an equivalent atom that has one of the following forms:

$$\begin{array}{l} t \leq p * x \\ t \leq 0 \\ t \leq -p * x \\ i \mid p * x + t \\ i \mid t \\ i \nmid p * x + t \\ i \nmid t \end{array}$$

where t has no occurrences of x, and p, q range (both here and for the rest of the section) over positive integers. We can ensure that the x-coefficients of (non)divisibility atoms are positive since multiplying their right-hand sides by -1 does not affect their truth values. Let ϕ_1 be the formula obtained by rewriting each atom of ϕ into one of these forms. ϕ and ϕ_1 are clearly equivalent, since all atom-pairs in corresponding positions of ϕ and ϕ_1 are equivalent. Let m be the least common multiple of all x-coefficients in ϕ_1, and Let ϕ_2 be the result of replacing each atom of ϕ_1 with the mapping

$$\begin{aligned} t \leq p * x &\mapsto (m/p) * t \leq m * x \\ t \leq -p * x &\mapsto (m/p) * t \leq -m * x \\ i \mid p * x + t &\mapsto (m/p) * i \mid m * x + (m/p) * t \\ i \nmid p * x + t &\mapsto (m/p) * i \nmid m * x + (m/p) * t \\ a &\mapsto a \end{aligned}$$

where a (both here and in other mappings in the section) matches all atoms not matched by preceding lines. The divisions and multiplications in $(m/p)*i$ or $(m/p) * t$ are not actual function symbols that appear in the new atoms (since that would not be generally well-formed), and we assume that they are eliminated by distribution and normalization. Intuitively, atoms without occurrences of x remain unchanged, and all coefficients of x are set to $\pm m$. Again, ϕ_1 is equivalent to ϕ_2, since each atom a in ϕ_1 is either unchanged, or all terms in a are multiplied by a positive integer.

Finally, let ϕ_3 be the formula obtained by replacing each coefficient of x in ϕ_2 with its sign, by applying the mapping

$$\begin{aligned} t \leq p * x &\mapsto t \leq 1 * x \\ t \leq -p * x &\mapsto t \leq -1 * x \\ i \mid p * x + t &\mapsto i \mid 1 * x + t \\ i \nmid p * x + t &\mapsto i \nmid 1 * x + t \\ a &\mapsto a \end{aligned}$$

Although ϕ_3 is not equivalent to ϕ_2, it satisfies a more limited kind of equivalence

$$\exists x.(m \mid 1 * x + 0 \wedge \phi_3) \leftrightarrow \exists x.\phi_2$$

where m is the absolute value of all x-coefficients in ϕ_2. The equivalence can be shown by the following: if some k is a witness of $\exists x.(m \mid 1 * x + 0 \wedge \phi_3)$, then $\phi_3[x/k]$[2] holds, and also $m * j = k$ for some j since $m \mid k$ holds. Then j is a witness of $\exists x.\phi_2$, since $\phi_2[x/j] \leftrightarrow \phi_3[x/m * j]$ by definition of ϕ_3, and $m * j = k$. Conversely, if some j is a witness of $\exists x.\phi_2$, then $\phi_3[x/m * j]$ holds by definition of ϕ_3 and $m \mid m * j$ holds trivially. Therefore, $m * j$ is a witness of $\exists x.(m \mid 1 * x + 0 \wedge \phi_3)$. Putting everything together, we have

$$\exists x.\phi \leftrightarrow \exists x.\phi_1 \leftrightarrow \exists x.\phi_2 \leftrightarrow \exists x.(m \mid 1 * x + 0 \wedge \phi_3)$$

[2] $\phi[x/t]$ is the result of replacing all free occurrences of x in ϕ with t.

where the first two equivalences hold by equivalence of quantified bodies of formulas. Also, $m \mid 1 * x + 0 \wedge \phi_3$ is x-normal by definition of ϕ_3. Therefore, $m \mid 1 * x + 0 \wedge \phi_3$ is the formula which satisfies all requisites. $\square$

Now it remains to show that there exists a QFE of $\exists x.\phi$ for any x-normal formula ϕ. The key insight for this part is a case analysis on the value of x. If $\exists x.\phi$ holds, it must fall into one of two cases: the *unbounded case* in which there are arbitrarily small values of x for which ϕ holds, and the *bounded case* in which there is a smallest value of x for which ϕ holds. If we can construct formulas ϕ_U and ϕ_B such that ϕ_U is implied by ϕ in the unbounded case and ϕ_B is implied by ϕ in the bounded case, then their disjunction $\phi_U \vee \phi_B$ is implied by $\exists x.\phi$. Furthermore, if ϕ_U and ϕ_B each independently imply $\exists x.\phi$, then $\phi_U \vee \phi_B$ is equivalent to $\exists x.\phi$. Finally, if ϕ_U and ϕ_B are quantifier-free, then $\phi_U \vee \phi_B$ is a QFE of $\exists x.\phi$. We show that formulas ϕ_U and ϕ_B which meet all these conditions can be constructed for any x-normal formula ϕ.

Definition 2.6. For any x-normal formula $\phi[x]$[3], $\phi_{-\infty}[x]$ is the result of replacing every atom of $\phi[x]$ with the mapping

$$\begin{aligned} t \leq 1 * x &\mapsto \bot \\ t \leq -1 * x &\mapsto \top \\ a &\mapsto a \end{aligned}$$

One useful way to think of $\phi_{-\infty}[x]$ is that it is the 'limit' of $\phi[x]$ as the value of x approaches negative infinity:

Theorem 2.7. *For any valuation v and x-normal formula $\phi[x]$, there exists an integer l such that for any integer $k < l$, $v\{x \mapsto k\}$[4] $\models \phi[x] \leftrightarrow \phi_{-\infty}[x]$.*

Proof. Observe that if the theorem holds in the limited case where $\phi[x]$ is atomic, then it holds for an arbitrary $\phi[x]$ by the following: first, apply the theorem to the atoms $a_1, ..., a_n$ of $\phi[x]$ to obtain the respective lower bounds $l_1, ..., l_n$, and set l to their minimum. Then for any $k < l$, k is smaller than the lower bound l_i of any atom a_i in $\phi[x]$. Therefore, each atom a_i in $\phi[x]$ is equivalent under $v\{x \mapsto k\}$ to the formula in the corresponding

[3]'$\phi[t]$' is a simplified version of the notation '$\phi[x/t]$'. The implicit variable x will always be clear from context.

[4]$v\{x \mapsto k\}$ is the valuation that is identical to v except that it maps the variable x to k.

position of $\phi_{-\infty}[x]$, and hence $\phi[x]$ and $\phi_{-\infty}[x]$ themselves are equivalent under $v\{x \mapsto k\}$.

For the case $\phi[x] = t \leq 1 * x$ (resp. $\phi[x] = t \leq -1 * x$), any value of x smaller than $[\![t]\!]_v$[5] (resp. $-[\![t]\!]_v$) will make $\phi[x]$ false (resp. true), and hence equivalent to $\bot$ (resp. $\top$). The remaining cases hold by reflexivity. $\square$

Note that $\phi_{-\infty}[x]$ already has one of the properties we require of ϕ_U, as it is implied by ϕ in the unbounded case. But it still includes free occurrences of x, which requires a few extra steps to remove.

Definition 2.7. An integer k is a **relevant divisor** of an x-normal formula $\phi[x]$ if an atom of the form $k \mid 1 * x + t$ or $k \nmid 1 * x + t$ occurs in $\phi[x]$. The **period** of $\phi[x]$ is the least common multiple of its relevant divisors.

The significance of periods is that the truth value of all (non)divisibility atoms in an x-normal formula is periodic in x with this value. This property is useful for proving the following:

Theorem 2.8. *If p is the period of an x-normal formula $\phi[x]$ and i, j are integers such that $i \equiv j \pmod{p}$, then $\models \phi_{-\infty}[i] \leftrightarrow \phi_{-\infty}[j]$.*

Proof. Observe that x only occurs in $\phi_{-\infty}[x]$ in (non)divisibility atoms of the form $k \mid 1 * x + t$ or $k \nmid 1 * x + t$. Since the mapping for obtaining $\phi_{-\infty}[x]$ from $\phi[x]$ does not modify (non)divisibility atoms, the divisor k in any such atom is a relevant divisor of $\phi[x]$, and hence divides p. Therefore, the truth value of any atom in $\phi_{-\infty}[x]$ is preserved by incrementing or decrementing the value of x by multiples of p. Since $i = j + m * p$ for some integer m, $\phi_{-\infty}[i]$ and $\phi_{-\infty}[j]$ are equivalent under any valuation. $\square$

Theorem 2.8 shows that the truth value of $\phi_{-\infty}[x]$ is periodic in x, which is the key to eliminating free occurrences of x.

Theorem 2.9. *If p is the period of an x-normal formula $\phi[x]$, then $\vDash \phi_{-\infty}[x] \rightarrow \bigvee_{0 \leq x < p} \phi_{-\infty}[x]$.*

Proof. If $v \models \phi_{-\infty}[x]$ for some v, then $v \vDash \phi_{-\infty}[i]$ for some integer i. Since $i \equiv i \bmod p \pmod{p}$, $\phi_{-\infty}[i]$ and $\phi_{-\infty}[i \bmod p]$ are equivalent under any valuation by Theorem 2.8, and hence $v \models \phi_{-\infty}[i \bmod p]$. It follows that one of the disjuncts of $\bigvee_{0 \leq x < p} \phi_{-\infty}[x]$ holds under v, since $0 \leq i \bmod p < p$. $\square$

[5] $[\![x]\!]_v$ is the value of x under valuation v. If x is a term, $[\![x]\!]_v$ is an integer; if it x is a formula, $[\![x]\!]_v$ is either true or false.

We now show that $\bigvee_{0\le x<p} \phi_{-\infty}[x]$ is the formula which satisfies the requisites of ϕ_U.

Theorem 2.10. *If p is the period of an x-normal formula $\phi[x]$ and v is an arbitrary valuation, the two statements*

1. *For any integer i, there exists an integer j such that $j < i$ and $v\{x \mapsto j\} \models \phi[x]$*
2. *$v \models \bigvee_{0\le x<p} \phi_{-\infty}[x]$*

are equivalent.

Proof. Suppose statement 1 holds. By theorem 2.7, there exists an l such that for any $i < l$, $\phi[x]$ and $\phi_{-\infty}[x]$ are equivalent under $v\{x \mapsto i\}$. Let k be an integer such that $v\{x \mapsto k\} \models \phi[x]$ and $k < l$, which exists by statement 1. Then we have $v\{x \mapsto k\} \models \phi_{-\infty}[x]$, which implies $v\{x \mapsto k\} \models \bigvee_{0\le x<p} \phi_{-\infty}[x]$ by Theorem 2.9. Since $\bigvee_{0\le x<p} \phi_{-\infty}[x]$ has no free occurrences of x, we have $v \models \bigvee_{0\le x<p} \phi_{-\infty}[x]$.

For the converse, suppose statement 2 is true. Again, let l be the lower bound which exists by Theorem 2.7, and i an arbitrary integer. It suffices to show that there exists an integer j such that $j < i$ and $v\{x \mapsto j\} \models \phi[x]$. By statement 2, there exists some integer k such that $v \models \phi_{-\infty}[k]$. Let j be an integer such that $j < l$, $j < i$, and $j = k \pmod p$. We know that such a j must exist, since p is nonzero. By Theorem 2.8 and $j = k \pmod p$, we have $v \models \phi_{-\infty}[j]$, and hence $v\{x \mapsto j\} \models \phi_{-\infty}[x]$. Since $j < l$, $v\{x \mapsto j\} \models \phi[x]$. □

For the bounded case and ϕ_B, we first need to define some auxiliary notions.

Definition 2.8. A term t is a **boundary point** of an x-normal formula $\phi[x]$ if $t \le 1 * x$ occurs in $\phi[x]$. The set of all boundary points of $\phi[x]$ is its **B-set**.

The significance of boundary points is that they are the values of x at which the truth value of $\phi[x]$ can change from true to false. This can be made more precise by the following:

Theorem 2.11. *For any valuation v, x-normal formula $\phi[x]$, integer k, and positive integer p such that the period of $\phi[x]$ divides p, if $v\{x \mapsto k\} \models \phi[x]$ but $v\{x \mapsto k-p\} \not\models \phi[x]$, then $[\![t]\!]_v \le k < [\![t]\!]_v + p$ for some t in the B-set of $\phi[x]$.*

Proof. By induction on the x-normal formula $\phi[x]$. There are two main cases to consider.

1. If $\phi[x] = \psi[x] \vee \chi[x]$, then either $\psi[x]$ or $\chi[x]$ is true under $v\{x \mapsto k\}$ but false under $v\{x \mapsto k-p\}$. Suppose $\psi[x]$ is (proof is symmetrical in the other case). Since any relevant divisor of $\psi[x]$ is also a relevant divisor of $\phi[x]$, the period of $\psi[x]$ divides p. By IH, there exists some t in the B-set of $\psi[x]$ such that $[\![t]\!]_v \leq k < [\![t]\!]_v + p$, and t is also in the B-set of $\phi[x]$ by definition of B-sets.

2. If $\phi[x] = t \leq 1 * x$, then we have $[\![t]\!]_v \leq k < [\![t]\!]_v + p$ from $v\{x \mapsto k\} \models t \leq 1 * x$ and $v\{x \mapsto k-p\} \not\models t \leq 1 * x$, and t is trivially in the B-set of $t \leq 1 * x$, the singleton $\{t\}$.

The case $\phi[x] = \psi[x] \wedge \chi[x]$ is similar to the case $\phi[x] = \psi[x] \vee \chi[x]$. In the remaining cases where $\phi[x]$ is atomic, $\phi[x]$ either has no occurrences of x, or has the form $t \leq -1 * x$, $i \mid 1 * x + t$, or $i \nmid 1 * x + t$, where the divisor i divides p. In each of these cases, the truth value of $\phi[x]$ cannot change from true to false when the value of x decreases by p, so the conclusion holds vacuously. □

Using B-sets, we can now construct a disjunction that satisfies the requisites of ϕ_B.

Theorem 2.12. *For any valuation v, x-normal formula $\phi[x]$, and integer k such that $v\{x \mapsto k\} \models \phi[x]$, if $v\{x \mapsto i\} \not\models \phi[x]$ for any $i < k$, then $v \models \bigvee_{b \in B} \bigvee_{0 \leq x < p} \phi[b+x]$, where B is the B-set of $\phi[x]$ and p is the period of $\phi[x]$.*

Proof. By assumption, $v\{x \mapsto k-p\} \not\models \phi[x]$. By theorem 2.11, $[\![t]\!]_v \leq k < [\![t]\!]_v + p$ for some $t \in B$. In other words, there exists some $0 \leq i < p$ such that $k = [\![t+i]\!]_v$. By substitution, $v\{x \mapsto [\![t+i]\!]_v\} \models \phi[x]$, which is equivalent to $v \models \phi[t+i]$. Since $t \in B$ and $0 \leq i < p$, we have $v \models \bigvee_{b \in B} \bigvee_{0 \leq x < p} \phi[b+x]$. □

Finally, the main theorem of the section:

Theorem 2.13. *For any x-normal formula $\phi[x]$, if B is the B-set of $\phi[x]$ and p its period, then*

$$\models \exists x.\phi[x] \leftrightarrow \left(\left(\bigvee_{0 \leq x < p} \phi_{-\infty}[x] \right) \vee \left(\bigvee_{b \in B} \bigvee_{0 \leq x < p} \phi[b+x] \right) \right)$$

Proof. We show that the equivalence holds under an arbitrary valuation v. If $v \models \exists x.\phi[x]$, then one of the following must hold: either (1) for any integer i, there exists a $j < i$ such that $v\{x \mapsto j\} \models \phi[x]$, or (2) there exists a smallest integer i such that $v\{x \mapsto i\} \models \phi[x]$. (1) and (2) each imply $v \models \bigvee_{0 \leq x < p} \phi_{-\infty}[x]$ and $v \models \bigvee_{b \in B} \bigvee_{0 \leq x < p} \phi[b + x]$ by theorems 2.10 and 2.12, respectively.

In the converse direction, if $v \models \bigvee_{0 \leq x < p} \phi_{-\infty}[x]$, then there exists an integer k such that $v\{x \mapsto k\} \models \phi[x]$ by Theorem 2.10, so $\exists x.\phi[x]$ holds under v with the witness k. If $v \models \bigvee_{b \in B} \bigvee_{0 \leq x < p} \phi[b + x]$, then there exists some $t \in B$ and $0 \leq i < p$ such that $v \models \phi[t + i]$. Since this is equivalent to $v\{x \mapsto [\![t + i]\!]_v\} \models \phi[x]$, $\exists x.\phi[x]$ holds under v with the witness $[\![t + i]\!]_v$. $\square$

It is also clear from its construction that

$$(\bigvee_{0 \leq x < p} \phi_{-\infty}[x]) \vee (\bigvee_{b \in B} \bigvee_{0 \leq x < p} \phi[b + x])$$

has no quantifiers provided that $\phi[x]$ is quantifier free, so Theorem 2.13 implies that for any x-normal formula $\phi[x]$, there exists a QFE of $\exists x.\phi[x]$. Together with Theorem 2.6, this is sufficient to show that whenever $\phi[x]$ is free of negations and quantifiers, a QFE of $\exists x.\phi[x]$ exists.

2.3 Computational Reflection

This section provides an overview of computational reflection, mostly following the presentation of Boutin [2].

A computational reflection procedure consists of the following:

- A **target syntax** α, the set of terms to be rewritten by reflection. Note that by 'terms' we do not mean first-order terms, but terms of the theorem prover's object language. Therefore, α may or may not be a subset of the terms of the object language that encode propositions.

- **Shadow syntaxes** $\beta_1, ..., \beta_{k+1}$, the sets of terms into which elements of α are reflected and manipulated.

- An **evaluation function** $V_i : \beta_i \to \alpha$ for each $1 \leq i \leq k$, defined in the object language, which provides the semantics for β_i in terms of α. In other words, each term $b_i \in \beta_i$ can be thought of as 'denoting' the term $V_i(b_i) \in \alpha$. To emphasize this relationship, we say that $b_i \in \beta_i$

is a **reflection** of $a \in \alpha$ in β_i, and that a is the **denotation** of b_i, if $V_i(b_i) = a$ holds.

- A **normalization function** $N_i : \beta_i \to \beta_{i+1}$ for each $1 \leq i \leq k$, also defined in the object language, which satisfies

  $$\vdash \forall b_i \in \beta_i.\ V_{i+1}(N_i(b_i)) = V_i(b_i)$$

  where '$\vdash$' denotes provability in the theorem prover. This is the function which performs the desired rewriting.

- A **reification function** which, for any given $a \in \alpha$, generates a corresponding $\ulcorner a \urcorner \in \beta_1$ such that $\vdash V_1(\ulcorner a \urcorner) = a$. Note that it may not be possible to define the reification function in the object language, in which case it must be implemented in the metalanguage instead.

In the simplest case, $k = 1$ and $\beta_1 = \beta_2 = \beta$ for a single shadow syntax β. Given these components and a goal $\phi[a]$ with occurrences of a term $a : \alpha$ to be rewritten, computational reflection can be carried out in the following steps:

1. Call the reification function with a to generate its reflection $\ulcorner a \urcorner \in \beta_1$.
2. Use $\ulcorner a \urcorner$ to state and prove $V_1(\ulcorner a \urcorner) = a$.
3. Rewrite the goal to $\phi[V_1(\ulcorner a \urcorner)]$ using the equality proven in step 2.
4. For each $1 \leq i \leq k$, do: let $\phi[V_i(t)]$ be the current goal. Instantiate the correctness lemma $\forall b_i \in \beta_i.\ V_{i+1}(N_i(b_i)) = V_i(b_i)$ to $V_{i+1}(N_i(t)) = V_i(t)$, and rewrite the goal with instantiated equality to $\phi[V_{i+1}(N_i(t))]$.

After the final iteration at $i = k$, the goal is transformed to the form $\phi[V_{k+1}(N_k(...N_1(\ulcorner a \urcorner)...))]$. The premise is that $V_{k+1}(N_k(...N_1(\ulcorner a \urcorner)...))$ normalizes to some $a' \in \alpha$ that you want to replace a with, which allows you to obtain the new goal $\phi[a']$. Precisely how that normalization takes place depends on the details of the theorem prover. In provers based on dependent type theory with an inherent notion of computation, you may simply unfold the definitions of $N_1, ..., N_k$ and V_{k+1} to obtain the new goal $\phi[a']$ directly in the proof state. In other provers, you may have to resort to indirect methods, such as proving $V_{k+1}(N_k(...N_1(\ulcorner a \urcorner)...)) = a'$ as a separate lemma or using code extraction.

For illustration, let us consider a concrete example. Suppose you want a tactic for normalizing Lean terms of the form `m*n:nat`, where `m:nat` and `n:nat` are very large. Naively normalizing `m*n` by unfolding the definition of `nat` multiplication is highly inefficient because `nat` is defined as unary numbers. One possible solution is to reflect `m` and `n` into the type `num` (a type which is isomorphic to `nat`, but much more efficient for computation due to its binary representation), normalize the product, and reflect the result back to `nat`. Following the scheme given above, we can obtain a computational reflection tactic by setting

$$\begin{aligned}
\alpha &:= \{\ \texttt{m*n} \mid \texttt{m:nat}, \texttt{n:nat}\ \} \\
\beta_1 &:= \texttt{num}\times\texttt{num} \\
\beta_2 &:= \texttt{num} \\
V_1 &:= \texttt{λ(x:num}\times\texttt{num),(}\uparrow\texttt{x.fst:nat)*(}\uparrow\texttt{x.snd:nat)}^{6} \\
V_2 &:= \texttt{λ(x:num),(}\uparrow\texttt{x:nat)} \\
N_1 &:= \texttt{λ(x:num}\times\texttt{num),x.fst*x.snd} \\
\ulcorner\texttt{m*n}\urcorner &:= \texttt{((}\uparrow\texttt{x:num),(}\uparrow\texttt{y:num))}
\end{aligned}$$

There is a slight abuse of notation in the definitions above as we supply Lean types and functions to fields that expect abstract sets and functions, but the intended meaning is clear. Using this setup and following the steps given above, any `m*n:nat` can be rewritten into `↑((↑m:num)*(↑n:num)):nat`, and the normalization of this new term is more efficient than that of `m*n:nat` as intended.

All this may seem like much ado for nothing, considering that we can simply prove `∀x:nat,∀y:nat,x*y=(↑((↑x:num)*(↑y:num)):nat)` and rewrite with it to the same effect. But notice that this simpler solution works only because we have chosen a restrictive α whose elements have only one possible form of `m*n`. Suppose α was defined more liberally as, say, the set of all terms that can be constructed with `nat` and multiplication (e.g., `(k*(m*n))*m` $\in \alpha$). Then computational reflection would still work provided that β_1 is set to a new inductive type isomorphic to this new target syntax, whereas it would require an infinite number of lemmas to achieve the same effect with simple rewriting.

[6] The upward arrow ↑ is Lean's notation for coercion: when Lean sees a term `↑n:nat` for some `n:num`, Lean uses the types to figure out that a `num` is being coerced into a `nat`, finds the instance of coercion from `num` to `nat` in the environment, and applies it to `n` to obtain its corresponding `nat`.

Also, note that reflected decision procedures in Lean can be obtained as a special case of computational reflection by setting

$$\begin{aligned}
\alpha &:= \{\ \texttt{P} \mid \texttt{P:Prop} \text{ denotes a formula in } L\} \\
\beta_i &:= \text{An inductive type } \texttt{B\_i} \text{ which encodes } L \\
V_i &:= \text{A function } \texttt{V\_i:B\_i}\to\texttt{Prop} \text{ which sends any } \texttt{b\_i:B\_i} \\
&\qquad \text{to a } \texttt{Prop} \text{ which denotes a formula in } L \\
N_i &:= \text{A function } \texttt{N\_i:B\_i}\to\texttt{B\_Si} \text{ that performs some part of} \\
&\qquad \text{normalization to decidable formulas} \\
\ulcorner\texttt{P}\urcorner &:= \texttt{b\_1:B\_1}, \text{ where } \texttt{b\_1} \text{ satisfies } \texttt{V\_1 b\_1 = P}
\end{aligned}$$

Again, $k+1$ is the number of shadow syntaxes, i ranges over $0 \leq i \leq k$, and L is the language of the structure decided by the procedure. Then for any goal `P:Prop` which denotes a formula in L, computational reflection replaces it with a new goal `V_Sk (N_k (... N_1(b_1) ...))` for some `b_1:B_1`, and normalizing this goal yields a decidable `Prop`. Quantifier-elimination based decision procedures, in turn, are special cases where the functions `N_1`,...,`N_k` removes all quantifiers from an input.

At a cursory glance, computational reflection can seem a convoluted way of rewriting terms. Why go through the trouble of defining new types and reflecting terms between them? The foremost reason is performance. In some cases, it is actually possible to replicate everything that computational reflection does with just tactical rewriting (i.e., by rewriting $\phi[a_1]$ with lemmas $a_1 = a_2,\ a_2 = a_3\ ...\ a_{n-1} = a_n$ to obtain $\phi[a_n]$), but this generally results in an undesirably large proof term. If a theorem prover is based on first-order logic, it implicitly applies the rule

$$\frac{\Gamma \vdash \phi[a_1] \qquad \Delta \vdash a_1 = a_2}{\Gamma, \Delta \vdash \phi[a_2]}$$

every time you rewrite a goal $\phi[a_1]$ with an equality $a_1 = a_2$, and a more complex analogue of the rule if it is based on a richer calculus. In other words, the proof term grows by *at least* the size of formula ϕ with every rewrite. If we assume that the number of rewrites is linear with the size of the goal, then the computational complexity of the whole procedure is $O(n^2)$. Computational reflection, on the other hand, always performs a fixed number of rewrites, and relegates rest of the work to term evaluation.

The second important consideration is control. Even if computational reflection is superior to tactical rewriting, there is no need to complicate things with shadow syntaxes if a normalization function N_1 can be defined directly from α to α. But there are many cases in which this is not possible. One obvious example is quantifier elimination — it's impossible to define a quantifier elimination function `qe:Prop→Prop` in, say, Lean or Coq, because the type of propositions is not an inductive type and has no recursion principle, which prevents the use of any control statements in `qe` that depend on the shape of the input formula. The toy tactic we discussed above for normalizing products of natural numbers is another example that suffers from the same problem: we cannot define a function `norm:nat→nat` which, given an argument `m*n:nat`, exploits the fact that it is a product of `m` and `n` and normalizes it in an intelligent way, because the only case analysis we get from the two constructors of `nat` is between 0 and $n + 1$. In other words, computational reflection is useful because reflecting a term into the shadow syntax equipped with the *right kind of constructors* gives you the ability to access and manipulate the term's internal structure.

3 Implementation

In this section, we discuss how decision procedures can be implemented in Lean by computational reflection, using the tactic `lia` as a concrete example. The following subsections assume that you know what a tactic monad is and can read basic tactic notations, but are otherwise self-contained.

One aspect of implementation which we will *not* discuss in detail is the formal proof: the largest part of the codebase of a typical reflection tactic will be taken up by the correctness proof of its normalization functions, but such proofs are not interesting since they stay entirely within the object language and are just like any other proof in the given theorem prover. Therefore, we focus on how to implement a working tactic *given* that the key correctness lemmas are already proven.

3.1 Planning

Recall from Section 2.3 that computational reflection consists of five main components: target syntax, shadow syntaxes, evaluation functions, normalization functions, and reification function. The first step in implementing a reflected decision procedure in a theorem prover is choosing the datatypes and functions that you will use for each of these components. These datatypes and functions don't have to be concretely defined yet; all you need is a short description of their requirements, which serves as a high-level plan for your implementation. For instance, what components do we need for implementing Cooper's algorithm in Lean? One intuitive answer would be:

$$\begin{aligned}
\alpha &:= \{\ \texttt{P} \mid \texttt{P:Prop} \text{ denotes a formula in } L_{LIA}\} \\
\beta_1 &:= \text{An inductive type } \texttt{form} \text{ which encodes } L_{LIA} \\
&\quad\ \text{(i.e., whose constructors correspond to the} \\
&\quad\ \text{function and predicate symbols of } L_{LIA}) \\
V_1 &:= \text{A function } \texttt{form.eval:form} \rightarrow \texttt{Prop} \text{ which} \\
&\quad\ \text{maps any } \texttt{f:form} \text{ to its corresponding } \texttt{Prop} \\
N_1 &:= \text{A function } \texttt{qe:form} \rightarrow \texttt{form} \text{ that eliminates} \\
&\quad\ \text{quantifiers using Cooper's algorithm} \\
\ulcorner \texttt{P} \urcorner &:= \texttt{f:form}\text{, where } \texttt{f} \text{ satisfies } \texttt{f.eval = P}^{7}
\end{aligned}$$

But this setup is impractical, because L_{LIA} imposes too rigid a restriction on how its terms can be constructed. For instance, $-1 * x$ is a well-formed term of L_{LIA}, but $-x$ is not; and we wouldn't want a tactic that fails whenever the latter occurs in the goal. More generally, if L is the language on which a decision procedure is defined, the set of `Prop`s which denote formulas of L is usually *not* a good target syntax, because decision procedures are typically defined on highly restricted languages to simplify their correctness proofs. In many cases, it is more convenient to define and use a new language L_+ that makes some sensible extensions to L by adding syntactic sugars. For `lia`, we use the language L_{LIA+}, whose terms and atoms are defined as:

$$t := \ i \ \Big|\ x \ \Big|\ i * x \ \Big|\ x * i \ \Big|\ -t \ \Big|\ t + t \ \Big|\ t - t$$

$$a := \ t \leq t \ \Big|\ i \mid t$$

where t ranges over terms, i over integer constants, x over integer variables, and a over atoms. Note that L_{LIA+} does not include the predicate symbol $\nmid$; since Lean's core library does not include a nondivisibility predicate, there is

[7] For any terms `a:A` and `A.f:A→B`, the notation `a.f` is a shorthand for `A.f a:B`.

no need to include a corresponding symbol in a language designed for accommodating Lean goals. There is a slight problem with using the set of `Prop`s that denote formulas of L_{LIA+} as the target syntax, since Cooper's algorithm is not defined to work with its additional function symbols. This problem can be solved by adding a normalization step that removes all syntactic sugars from a formula of L_{LIA+} and returns an equivalent formula of L_{LIA}. The resulting setup is as follows:

α := { `P` | `P:Prop` denotes a formula in L_{LIA+}}
β_1 := An inductive type `preform` which encodes L_{LIA+}
β_2 := An inductive type `form` which encodes L_{LIA}
V_1 := A function `preform.eval:preform→Prop` which maps any `p:preform` to its corresponding `Prop`
V_2 := A function `form.eval:form→Prop` which maps any `f:form` to its corresponding `Prop`
N_1 := A function `trim:preform→form` that removes the syntactic sugars of L_{LIA+}
N_2 := A function `qe:form→form` that eliminates quantifiers using Cooper's algorithm
⌜`P`⌝ := `p:preform`, where `p` satisfies `p.eval = P`

In fact, the principal reason for accommodating multiple shadow syntaxes in computational reflection is that it allows you to add this kind of preprocessing step whenever necessary.

Now that the overall plan is fixed, we can start writing some concrete Lean definitions. The definition of `preform` simply falls out of the definition of L_{LIA+}:

```
inductive preterm : Type
| const  : int → preterm
| var    : nat → preterm
| mulvar : int → nat → preterm
| varmul : nat → int → preterm
| neg    : preterm → preterm
| add    : preterm → preterm → preterm
| sub    : preterm → preterm → preterm
```

```
inductive preform : Type
| le    : preterm → preterm → preform
| dvd   : int → preterm → preform
| true  : preform
| false : preform
| not   : preform → preform
| or    : preform → preform → preform
| and   : preform → preform → preform
| imp   : preform → preform → preform
| iff   : preform → preform → preform
| fa    : preform → preform
| ex    : preform → preform
```

The definition of `preterm` matches the definition of terms of L_{LIA+} exactly, constructor-by-constructor, in the same order. Note that the definition uses de Brujin indices, which means that each variable is represented by a `nat`. For instance, the terms `preterm.mulvar i n` and `preterm.varmul n i` denote terms $i*x_n$ and x_n*i, where the variable x_n is bound by the $(n+1)$th quantifier encountered when traversing up the syntax tree from the variable. The constructors `le, dvd, true, false, not, or, and, imp, iff, fa, ex` encode the predicates and logical constants $\leq, |, \top, \bot, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists$, respectively. The definition of `preform.eval`, which maps each constructor to its corresponding function or predicate, is also straightforward:

```
def preterm.eval (l : list int) : preterm → int
| (preterm.const i)    := i
| (preterm.var n)      := l.inth n
| (preterm.mulvar i n) := i * (l.inth n)
| (preterm.varmul n i) := (l.inth n) * i
| (preterm.neg t)      := -t.eval
| (preterm.add t1 t2)  := t1.eval + t2.eval
| (preterm.sub t1 t2)  := t1.eval - t2.eval

def preform.eval : list int → preform → Prop
| l (preform.le t1 t2) := t1.eval l ≤ t2.eval l
| l (preform.dvd i t)  := i | t.eval l
| l preform.true       := _root_.true[8]
| l preform.false      := _root_.false
| l (preform.not p)    := ¬(p.eval l)
| l (preform.or p q)   := (p.eval l) ∨ (q.eval l)
| l (preform.and p q)  := (p.eval l) ∧ (q.eval l)
| l (preform.imp p q)  := (p.eval l) → (q.eval l)
| l (preform.iff p q)  := (p.eval l) ↔ (q.eval l)
| l (preform.fa p)     := ∀ x, p.eval (x::l)
| l (preform.ex p)     := ∃ x, p.eval (x::l)
```

Note that the valuation of variables is given as a list of integers, which is sufficient since each variable x is represented by a de Brujin index n, so you can just look up the position n in the list to obtain the value of x. The only nontrivial part is the evaluation of quantified formulas. Given a `preform.ex p:preform`, it is clear that the result of evaluating it to `Prop` must have the form `∃ x, P`, but it is not obvious how we can define the quantified body `P` to correctly include occurrences of the bound variable `x`. Passing valuations encoded as lists as a side argument and updating them between recursive calls is a clever solution used in Nipkow [8] which works well with de Brujin indices: by adding `x` to the head of the list, the values of all other variables are appropriately shifted by one position to accommodate the addition of a quantifier, and the new `x` is correctly bound by the closest quantifier.

The definition of `form` is a bit less intuitive than that of `preform`:

```
inductive form : Type
| le    : int → list int → form
| dvd   : int → int → list int → form
| ndvd  : int → int → list int → form
| true  : form
| false : form
| not   : form → form
| or    : form → form → form
| and   : form → form → form
| ex    : form → form
```

The constructors `le`, `dvd`, and `ndvd` encode atomic formulas of L_{LIA} in the

[8] `_root_` is used to avoid ambiguities due to namespaces. Here, `_root_.true` unambiguously refers to `true:Prop` instead of `preform.true:preform`.

following normal forms:

$$\begin{aligned} i &\leq c_0 * x_0 + ... + c_n * x_n \\ i &\mid j + c_0 * x_0 + ... + c_n * x_n \\ i &\nmid j + c_0 * x_0 + ... + c_n * x_n \end{aligned}$$

Observe that any inequality atom of the form $i \leq c_0 * x_0 + ... + c_n * x_n$ can be completely characterized by a single integer i and a list of integers $c_0, ..., c_n$, which is why the constructor `le` takes an integer and a list of integers as arguments. Similarly, constructors `dvd` and `ndvd` take two integers and one list of integers as arguments, which is sufficient for uniquely characterizing any (non)divisibility atom. The main advantage of imposing these normal forms on atoms is that it helps with the definition and correctness proof of `qe`, the quantifier elimination function. As we have seen in Section 2.2, many steps in Cooper's algorithm involve performing calculations with (or making changes to) the coefficient of the head variable (the variable to be eliminated) in atoms, and exposing this coefficient at the head of a list considerably simplifies operations on it. Also, notice that `form` does not have constructors for implications, biconditionals, or universal quantifiers. This does not affect the expressiveness of `form` since these logical constants can be eliminated using the equivalences

$$\begin{aligned} P \rightarrow Q &\iff \neg P \vee Q \\ P \leftrightarrow Q &\iff (P \wedge Q) \vee (\neg P \wedge \neg Q) \\ \forall x.P &\iff \neg \exists x.\neg P \end{aligned}$$

and the absence of these constants makes it easier to define and verify `qe` by reducing the number of cases to consider.

The evaluation function of `form` is very similar to that of `preform`:

```
def form.eval : list int → form → Prop
| l (form.le i is)      := i ≤ l · is
| l (form.dvd i j is)   := i ∣ j + l · is
| l (form.ndvd i j is) := ¬(i ∣ j + l · is)
| l form.true           := _root_.true
| l form.false          := _root_.false
| l (form.not f)        := ¬(f.eval l)
| l (form.or f g)       := (f.eval l) ∨ (g.eval l)
| l (form.and f g)      := (f.eval l) ∧ (g.eval l)
| l (form.ex f)         := ∃ x, f.eval (x::l)
```

where `l · is` denotes the dot product of `l` and `is`, defined by

```
def dot_prod : list int → list int → int
| [] []                   := 0
| [] (_::_)               := 0
| (_::_) []               := 0
| (i1::is1) (i2::is2) := (i1 * i2) + dot_prod is1 is2
infix `·` := dot_prod
```

The main takeaway from the contrast between `preform` and `form` is their specialization for respective purposes: `preform` faithfully mirrors the structure of input formulas in order to allow `lia` work with a wider range of goals, whereas `form` is optimized for performance and easy verification.

We omit the definition of normalization functions `trim:preform→form` and `qe:form→form`, since `trim` involves tedious algebraic manipulations and we have already discussed the algorithm of `qe` in Section 2.2. It suffices to state their main correctness theorems:

```
theorem eval_trim (l : list int) (p : preform) :
  (trim p).eval l ↔ p.eval l

theorem eval_qe (l : list int) (f : form) :
  (qe f).eval l ↔ f.eval l
```

The only remaining component is the reification function, which we treat separately in the next section.

3.2 Reification

Once we have fixed the target syntax α, the first shadow syntax β_1, and its evaluation function V_1, the next step is implementing reification – i.e., writing a tactic which replaces a given goal $\phi \in \alpha$ with its equivalent $V_1(\ulcorner\phi\urcorner)$, where $\ulcorner\phi\urcorner \in \beta_1$ is the reflection of ϕ in β_1. The hardest part in this step is stating the correct subgoal $V_1(\ulcorner\phi\urcorner) = \phi$; once this subgoal is stated, discharging it by reflexivity and rewriting the goal with it is trivial. The simplest solution, of course, is to prove the lemma

$$\forall a \in \alpha.\ V_1(\ulcorner a\urcorner) = a$$

in the theorem prover, which would allow us to obtain the correct subgoal by unifying a with the goal ϕ. But this is not generally possible, since the reification function $\ulcorner . \urcorner$ is not always definable in the theorem prover's object language. To put it in informal terms, there are cases in which we cannot state the theorem 'for any a, evaluating result of reifying a is equal to a', because there is no way to express 'the result of reifying a.' In particular, this is not possible for `lia` because we cannot (for reasons discussed in section 2.3) define a Lean function `reify:Prop→preform` that generates the right `p:preform` for any given `P:Prop`.

For decision procedures in Lean, one possible solution to this problem is to use the `expr` type. We can think of `expr` as the shadow syntax of Lean's object language, in the sense that Lean terms can be reified into its reflection in `expr`, and `expr`s can be evaluated to obtain the Lean terms they denote. The type `expr` is important for our purposes because a `tx:expr` which denotes a term `t:T` encodes all relevant information about the term `t`. Therefore, if we can't perform recursion on an arbitrary goal `P:Prop`, we can still achieve the same effect by recursion on the `Px:expr` which denotes `P`.

Here's a series of steps that uses `expr` to perform reification for `lia`, in informal terms:

1. Obtain `Px:expr`, the reflection of the goal `P:Prop` in `expr`.

2. Recurse on the structure of `Px` to obtain `px:expr`, the reflection of `p:preform` in expr, which in turn is the reflection of `P:Prop` in `preform`.

3. Use `px` and `Px` to assert and prove `p.eval [] ↔ P`[9]

4. Rewrite the goal `P` into a new goal `p.eval []` using the equivalence proven in step 3.

The Lean tactic `reify` which carries out these steps is defined as:

```
meta[10] def reify : tactic unit :=
do Px ← target,
   px ← expr.to_preform Px,
   apply (reify_aux px Px), skip
```

[9]We use the empty list here since it does not matter which list is used. Recall that the list argument to `preform.eval` provides the valuation for free variables. Since Cooper's algorithm only works for sentences, we assume that `lia` is only called with sentential goals, which means that the reflections of goals in `preform` are also sentences whose truth values are unaffected by changes in variable valuations.

tactic.target[11] is a tactic which returns the reflection of the current goal in expr, so the first line Px ← target corresponds to step 1 where Px is bound to the reflection of P in expr. The second line performs step 2, where the tactic expr.to_preform is defined as

```
meta def expr.to_preterm : expr → tactic expr
| (var n)              := return (app `(preterm.var) `(n))
| `(%%(var n) * %%ix) :=
  return (app (app `(preterm.varmul) `(n)) ix)
| `(%%ix * %%(var n)) := ...
| `(-%%tx)             := ...
| `(%%t1x + %%t2x)     :=
  do pt1x ← t1x.to_preterm,
     pt2x ← t2x.to_preterm,
     return (app (app `(preterm.add) pt1x) pt2x)
| `(%%t1x - %%t2x)     := ...
| ix                   := return (app `(preterm.const) ix)

meta def expr.to_preform : expr → tactic expr
| `(%%t1x ≤ %%t2x) :=
  do pt1x ← t1x.to_preterm,
     pt2x ← t2x.to_preterm,
     return (app (app `(preform.le) pt1x) pt2x)
| `(%%ix ∣ %%tx) := ...
| `(true) := return `(preform.true)
| `(false) := ...
| `(¬%%Px) := ...
| `(%%Px ∨ %%Qx) :=
  do px ← Px.to_preform,
     qx ← Qx.to_preform,
     return (app (app `(preform.or) px) qx)
| `(%%Px ∧ %%Qx) := ...
| `(%%Px ↔ %%Qx) := ...
```

[10]The keyword meta is used to mark definitions of *metaprograms*. Metaprograms differ from regular Lean programs in important ways, including use of arbitrary recursive calls. For more details, see Avigad, de Moura, and Roesch [1].

[11]We assume that namespaces expr and tactic are open, and freely drop prefixes in the code examples.

```
| (pi _ _ `(int) Qx) :=
  do qx ← Qx.to_preform,
     return (app `(preform.fa) qx)
| `(%%Px → %%Qx) :=
  do px ← Px.to_preform,
     qx ← (Qx.lower_vars 1 1).to_preform,
     return (app (app `(preform.imp) px) qx)
| `(Exists %%(lam _ _ _ Px)) :=
  do px ← Px.to_preform,
     return (app `(preform.ex) px)
| `(Exists %%Prx) :=
  do px ← (app (Prx.lift_vars 0 1) (var 0)).to_preform,
     return (app `(preform.ex) px)
| _ := failed
```

(Note that several match cases have their bodies abbreviated to . . ., because they are very similar to that of other cases and hence redundant.) Parts of the definition might not make sense if you're not used to working with `expr`s, so we'll explain them one by one. As we mentioned in earlier discussions, `expr.to_preform` works by indirect recursion on `expr`. In other words, instead of recursing on a `Prop` and returning a corresponding `preform`, it recurses on the `expr` of a `Prop` and returns the `expr` of the corresponding `preform`. This introduces some complications for pattern matching, because `expr` does not have the right kind of constructors for our purposes. For instance, the intuitive definition of `expr.to_preform` for the match case where the argument is a disjunction would be:

```
| (Px ∨ Qx) :=
  do px ← Px.to_preform,
     qx ← Qx.to_preform,
     ...
```

where you would then construct the return value using `px` and `qx`. But this is not possible, since ∨ is not a constructor of `expr`. To work around this problem, we use two features of Lean metaprogramming called *quotation* and *antiquotation*:

- For any term `t:T`, the term `` `(t):expr `` is the reflection of `t` in `expr`. In other words, the denotation of `` `(t) `` is `t`. `` `(t) `` is called the quotation of `t`.

- For any `x:expr` such that `x` denotes some `t:T`, appending `%%` to `x` gives back the denotation of `x`, such that `%%x = t`. `%%x` is called the antiquotation of `x`. Note that antiquotations can only be used inside quotations.

Quotations and antiquotations enable a much more intuitive kind of pattern matching on `expr`s; instead of dealing with the low-level details of the inductive definition of `expr`, we can case-analyze according to what each `expr` *denotes.* The disjunction match case in `expr.to_preform`, for instance, can be defined as

```
| `(%%Px ∨ %%Qx) :=
  do px ← Px.to_preform,
     qx ← Qx.to_preform,
     ...
```

Let's take a close look at what happens in this case. Suppose `expr.to_preform` is applied to the argument `x:expr`, where `x` denotes the proposition `P ∨ Q`. When the pattern `` `(%%Px ∨ %%Qx) `` is unified with `x`, their denotations must agree after unification, so `%%Px` and `%%Qx` are unified with propositions `P` and `Q`, respectively. By definition of antiquotations, this means that the `expr`s `Px` and `Qx` denote `P` and `Q`, respectively. Therefore, `Px` and `Qx` are precisely the arguments we need for recursive calls to `expr.to_preform`.

We also need to pay attention to how the return value is constructed, since the return type is `expr` and not `preform`. `expr.app` is the constructor of `expr` which encodes function application. In other words, if `fx:expr` denotes `f:A→B` and `ax:expr` denotes `a:A`, then `app fx ax:expr` denotes `f a:B`. Similarly, ``app (app `(preform.or) px) qx`` denotes `preform.or p q`, provided that `px` and `qx` denote `p:preform` and `q:preform`.

The match cases for quantifiers are a bit more complicated due to a number of subtle pitfalls. Notice, for instance, that the match cases are out of their usual order and places the case for universal quantifier (with the constructor `expr.pi`) before that of implication. Reification breaks down if this order is reversed, because both universal formulas and implications are instances of Π-types in Lean's dependent type theory (the proposition `P→Q` is just a shorthand for `Πx:P,Q`) and the pattern `` `(%%Px→%%Qx) `` will indiscriminately match both kinds of formulas. Also, the argument of `Exists` can be either a lambda term (e.g., `Exists (λx:int,0≤x)`) or a partially applied predicate (e.g., `Exists (int.le 0)`), so we need two separate match cases to accommodate them.

One of the principal reasons for using de Brujin indices in `preform` is that `expr` is also defined using them. This similarity helps reification (note the simplicity of the first match case of `expr.to_preterm`), but the downside of de Brujin indices is that you have to make minute modifications to make sure that they are always bound by the same quantifier. In the implication case of `expr.to_preform`, for instance, the indices in the consequent `Qx` have to be decremented using `expr.lower_vars` before recursive call because implication counts as a variable binder in `expr`, (due to the conflation of implications and universal quantifiers explained above), but not in `preform`. The second match case for existential formulas uses `expr.lift_vars` for similar reasons.

We now turn to the last line of `reify`, which carries out steps 3 and 4:

```
    apply (reify_aux px Px), skip
```

Before getting into details, let's think about how you'd perform steps 3 and 4 if you were in interactive mode instead of programming a tactic. Given that the goal is `P:Prop` and its reflection `p:preform` is available in the context, the following would do the trick:

```
  apply (@iff.elim_left (p.eval []) P (iff.refl _))
```

Therefore, it suffices to write a tactic that has the same effect as the interactive one-liner above. The main difference between the interactive and non-interactive versions of `apply` is that the latter requires an `expr` argument, so we need to construct an `expr` that denotes the term `@iff.elim_left (p.eval []) P (iff.refl _)`. This is precisely what `reify_aux` does, which is defined as:

```
  meta def reify_aux (px Px : expr) : expr :=
  app_of_list `(@iff.elim_left)
    [(app `(preform.eval []) px), Px, (app `(iff.refl) Px)]
```

The final `skip` does not do anything, and is only added to ensure that `reify` has the right type.

3.3 Discharging Goals

After reifying the goal and rewriting with correctness lemmas of normalizing functions, we obtain a goal of the form $V_{k+1}(N_k(...N_1(\ulcorner\phi\urcorner)...)$. In case of `lia`, this means we replace a goal `P:Prop` that denotes a formula of L_{LIA+}

with `(qe f).eval []` for some `f:form`. Since `qe f` is a QFE of `f`, and QFEs of sentences of L_{LIA+} are decidable, it should be possible to automatically discharge the goal `(qe f).eval []` provided that it is a true sentence of linear integer arithmetic. The actual implementation of this step, however, is subject to some important caveats.

The standard way to deal with decidable propositions in Lean is to use the class `decidable`, which is defined as:

```
class inductive decidable (p : Prop)
| is_false (h : ¬p) : decidable
| is_true  (h : p) : decidable
```

Intuitively, if you have a term `t:decidable P` for some `P:Prop`, you have a proof of either `P` or `¬P`. Since `decidable` is a class, this means that you can automatically obtain a proof of either `P` or `¬P` by typeclass inference, provided that an instance of type `decidable P` is available in the environment.

The easiest way to prove decidable and true propositions in Lean is to use the tactic `exact_dec_trivial`. Given a goal `P:Prop`, the tactic works by the following steps: first, it attempts to synthesize a term `t:decidable P` by typeclass inference. If it succeeds, it proceeds to pattern match on `t`. If `t` reduces to `is_false _`, the tactic fails; if `t` reduces to `is_true h` for some `h:P`, it applies `h` to close off the goal. Therefore, in order to use `exact_dec_trivial` for discharging goals of the form `(qe f).eval []`, Lean must be able to (1) synthesize `t:decidable ((qe f).eval [])` for any `f:form`, and (2) reduce the synthesized `t` to either `is_false h` or `is_true h` for some `h`. Condition (1) can be easily met by providing a few lemmas and instances. First, define a predicate `qfree` to express that a `form` is quantifier-free :

```
def qfree : form → Prop
| (form.not f)   := qfree f
| (form.or  f g) := qfree f ∧ qfree g
| (form.and f g) := qfree f ∧ qfree g
| (form.ex _)    := false
| _              := true
```

Then show that `qe f` is always quantifier-free, and also that the result of evaluating any quantifier-free `f:form` is decidable:

```
theorem qfree_qe : ∀ f : form, qfree (qe f)
theorem dec_eval_of_qfree : ∀ f : form, qfree f →
  ∀ l : list int, decidable (f.eval l)
```

This gives the instance that Lean can use to synthesize necessary decidability term for any `f`.

```
instance dec_eval_qe (f : form) (l : list int) :
  decidable ((qe f).eval l) :=
dec_eval_of_qfree _ (qfree_qe _) _
```

Condition (2) is more tricky to meet, because it concerns the evaluation behaviour of Lean which users have little direct control over. Although evaluation of any non-meta Lean term always terminates, the normal form of `t:decidable ((qe f).eval [])` is not guaranteed to be `is_false h` or `is_true h`. The main problem here is well-founded recursion, which Lean does not always know how to unfold correctly. For example, one of the earlier versions of `lia` had a bug which prevented automatic discharging of goals using `exact_dec_trivial`. The culprit was the NNF-transformation function used in `qe`, which was defined as

```
def nnf : form → form
...
| (form.not (form.not f))   := nnf f
| (form.not (form.or f g))  :=
  form.and (nnf f.not) (nnf g.not)
| (form.not (form.and f g)) :=
  form.or (nnf f.not) (nnf g.not)
...
```

(Match cases not relevant to the discussion are omitted.) Notice that `nnf` uses non-structural recursion (recursive calls with indirect subformulas) when the argument is a negation, which introduced well-founded recursion and caused the evaluation of decidability terms to prematurely terminate. Although such error may seem obvious in hindsight, it requires tedious code inspection to spot once it is introduced, and there seems to be no easy way to automate the debugging process (e.g., an option which displays all definitions in a file with well-founded recursion) at the moment. So the best one can do is to adhere to simple structural recursion whenever possible.

Once conditions (1) and (2) are satisfied, `exact_dec_trivial` can be used to automatically discharge quantifier-eliminated goals, but there still remain concerns regarding its performance. `exact_dec_trivial` relies on the kernel to evaluate decidability terms, which means it actually proves that the result of evaluation is equal to the original term. Although this is inevitable for proof synthesis and is the behaviour that you usually want, there may be exceptional circumstances where you must maximize performance at the expense of rigour. In such cases, we may opt to evaluate decidability terms via Lean's VM (virtual machine) instead. VM evaluation is much faster than kernel evaluation because it does not retain any information about types or proofs; the downside is that the user must trust the VM to correctly implements Lean's evaluation rules. The VM analogue of `exact_dec_trivial` is defined as follows:

```
meta def vm_dec_eval : tactic unit :=
do `((qe %%fx).eval []) ← target,
   f ← eval_expr form fx,
   match (dec_eval_qe f []) with
   | (is_true _) := admit
   | _           := failed
   end
```

Assuming that `vm_dec_eval` is called with the goal `(qe f).eval []`, the first line binds `fx:expr` to the `expr` which denotes `f`. The next line `f ← eval_expr form fx` makes `f` available inside the tactic. If `dec_eval_qe f []` evaluates to `is_true _`, the goal is closed off by `admit`, the tactic for admitting a goal without a proof term.[12] In the other case, the tactic simply fails. Although the way `vm_dec_eval` works is very similar to that of `exact_dec_trivial`, it is much faster because it is a metaprogram evaluated by the VM.

4 Test Results

When beginning work on this project, there was considerable skepticism regarding whether computational reflection is the right approach for automating linear integer arithmetic. The main concern was that evaluation of large

[12]The justification for using `admit` in this case is that, if a term `is_true h` has the type `decidable ((qe f).eval [])`, then the term `h` must have the type `(qe f).eval []`. This reasoning, of course, relies on the crucial assumption that the evaluation to `is_true h` was performed correctly by the VM.

goals with Lean's kernel might be impractically slow. In order to assess the feasibility of reflected decision procedures in Lean, the tactic `lia` was tested using a sample of 50 sentences of linear integer arithmetic. Some typical examples used in the test are as follows (see Appendix A for the full list):

```
ex02 : ∀ x y : int, (x ≤ 5 ∧ y ≤ 3) → x + y ≤ 8
ex06 : ∀ x : int, (x = -5 ∨ x = 7) → ¬x = 0
ex16 : ∀ x : int, (∃ y : int, 2 * y + 1 = x) →
       ¬∃ y : int, 4 * y = x
ex23 : ∀ x : int, (x < 43 ∧ x > 513) → ¬x = x
ex24 : ∀ x : int, ∃ y : int, x = 3 * y - 1 ∨
       x = 3 * y ∨ x = 3 * y + 1
```

Notice that the examples include symbols not in L_{LIA+}, such as $<, =$, and $>$. The version of `lia` used in the test included some minor optimizations on top of the baseline version whose code we discussed in Section 3, including a preprocessor that eliminates syntactic sugars and the use of more efficient datatypes. All tests were performed on an Intel Core i7-7500U CPU machine with 8 gigabytes of RAM. In a batch test using examples 1-34, `lia` took 12.536 seconds to discharge all goals, with an average of $\approx$ .36874 seconds per goal. Although it's hard to tell how this compares to other tactics without running a controlled experiment, it's fast enough to save time for users working with integer arithmetic, and surprising given the pessimistic initial expectations regarding kernel evaluation. Here's the detailed breakdown of execution time from Lean's profiler, with uninteresting entries omitted for legibility:

```
12536ms   100.0%   lia
 8801ms    70.2%   tactic.to_expr
 8771ms    70.0%   tactic.exact_dec_trivial._lambda_2
 8771ms    70.0%   tactic.interactive.exact
 3754ms    29.9%   reify
 3438ms    27.4%   reify._lambda_2
 3435ms    27.4%   tactic.try
 3435ms    27.4%   tactic.try_core
 3434ms    27.4%   tactic.interactive.propagate_tags
 3434ms    27.4%   tactic.interactive.simp_core
 3157ms    25.2%   tactic.interactive.simp_core_aux
 3156ms    25.2%   tactic.interactive.simp_core_aux._lambda_5
 3156ms    25.2%   tactic.simp_target
 3147ms    25.1%   tactic.simplify
```

The amount of time spent in `reify` is notable; before the test, it was expected that 90% or more of the total execution time would be spent in the kernel evaluation of `exact_dec_trivial`. In fact, the initial figures observed during early stages of development were close to 90%, but subsequent optimizations brought the percentage down to around 70. Furthermore, it seems that most of the reification time is taken up by the simplifier, which was used for elimination of syntactic sugars. This suggests that rewriting `reify` to be less reliant on the simplifier (e.g., by extending `preterm` and performing more syntactic sugar elimination with `trim`) may bring significant performance improvements.

For comparison, the same setup and examples were also used to test `lia_vm`, an alternative version of `lia` that is identical to `lia` in all respects except that it uses VM evaluation. Its total execution time clocked at 4.309 seconds, with the following breakdown:

```
 4309ms   100.0%   lia_vm
 3616ms    83.9%   reify
 3318ms    77.0%   reify._lambda_2
 3318ms    77.0%   tactic.try
 3318ms    77.0%   tactic.try_core
 3317ms    77.0%   tactic.interactive.propagate_tags
 3317ms    77.0%   tactic.interactive.simp_core
 3057ms    70.9%   tactic.interactive.simp_core_aux
 3056ms    70.9%   tactic.interactive.simp_core_aux._lambda_5
 3056ms    70.9%   tactic.simp_target
 3045ms    70.7%   tactic.simplify
  391ms     9.1%   vm_dec_eval
  355ms     8.2%   tactic.to_expr
  351ms     8.1%   dec_eval_qe
  285ms     6.6%   tactic.exact_dec_trivial._lambda_2
  285ms     6.6%   tactic.interactive.exact
```

The overall pattern in the result is as expected, with a steep decrease in time spent evaluating goals (`vm_dec_eval`). Although a threefold increase in performance is a disappointing payoff for giving up proof synthesis, this was expected since reification already took one third of the total time for `lia`, with which `lia_vm` shares the same `reify` tactic. If these results are representative, `lia_vm` would not be worth using for goals that `lia` proves under standard settings. At the same time, it suggests that the difference could be much more pronounced for harder goals, where the proportion of time spent in reification is smaller. In order to test this hypothesis, a separate test was performed using the harder examples 35-41, with the time limit increased to 1000000 (the default is 100000). In this test, `lia_vm` took a total of 55.406 seconds to prove all goals. The result for `lia`, however, was somewhat unexpected: it failed for all examples, but timed out for only one (36), and ran into either deep recursion or memory limit for all others. Therefore, the hypothesis regarding execution time could not be confirmed, but it still seems safe to say that there is a dramatic performance difference between `lia` and `lia_vm` for more difficult goals.

Finally, since `lia_vm` does not produce proofs and runs the risk of admitting false goals, a separate test was performed using the nontheorem examples 42-50. In this test, `lia_vm` failed as expected for all goals in 2.547 seconds.

5 Conclusion and Future Work

In this thesis, we outlined the main steps for implementing a decision procedure as a Lean tactic via computational reflection, and demonstrated that it is a viable approach to automation in Lean. One obvious direction in which we could extend this work is to implement more reflected decision procedures in Lean; since a large part of `lia`'s codebase can be reused for this purpose, it would take minimal effort to implement a tactic for, say, linear arithmetic over natural numbers.

Furthermore, the application of computation reflection is not limited to decision procedures: as we mentioned in Section 2.3, it can be used for virtually any kind of computational rewriting where the equivalence between the input and output can be proven. A potential alternative application would be a tactic that performs algebraic manipulations similar to computer algebra systems, for which there is high demand in the developer community working on Lean's standard math library.

Another related but more challenging goal would be to remove the limitations in the goal-discharging step. In the current version of `lia`, the only two options for closing off a quantifier-eliminated goal are either complete proof synthesis by kernel evaluation, or taking the result of VM evaluation on trust.

In many situations, it would be attractive to have a middle-ground solution that is faster than the kernel but provides at least a partial ground for trust – e.g., a verified evaluator like Coq's `vm_compute`. `lia` is also limited in that both the kernel and VM options for discharging goals are entirely automatic, which means it either succeeds completely or makes zero progress. If it were possible to step through intermediate stages of evaluation or switch to interactive mode after evaluating as much as possible, the tactic could offer useful information even in cases where it cannot prove a goal by itself.

6 Related Works

This thesis was most directly influenced by Nipkow [8], whose definitions and statements of theorems were ported to Lean and formed the basis of `lia`. This work is particularly interesting for its use of Isabelle's locales to factor out the common components of QEPs, which could save a significant amount of work when implementing multiple QEPs. The informal proofs in Section 2 and the formal correctness proofs of `lia` mostly follow those of Harrison [6]. Avigad et al. [1] is a comprehensive introduction to programming in Lean, including the tactic programming techniques used in Section 3.

Computational reflection was first proposed by Boyer and Moore [3]. Boutin [2] is an accessible introduction which gives intuitive explanations of the strengths of the technique. Empirical support for performance claims made in Section 2.3 can be found in Chaieb and Nipkow [4], which implements Cooper's algorithm in Isabelle/HOL twice, once in tactic style and and once via computational reflection, and shows the latter to be faster by 1-2 orders of magnitude.

7 References

[1] Jeremy Avigad, Leonardo de Moura, and Jared Roesch. Programming in lean, Dec 2016. `https://leanprover.github.io/programming_in_lean/programming_in_lean.pdf`.

[2] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *International Symposium on Theoretical Aspects of Computer Software*, pages 515–529. Springer, 1997.

[3] Robert S Boyer and J Strother Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. Technical report, SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, 1979.

[4] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41(1):33, 2008.

[5] David C Cooper. Theorem proving in arithmetic without multiplication. *Machine intelligence*, 7(91-99):300, 1972.

[6] John Harrison. *Handbook of practical logic and automated reasoning.* Cambridge University Press, 2009.

[7] Pattie Maes. Computational reflection. *The Knowledge Engineering Review*, 3(1):1–19, 1988.

[8] Tobias Nipkow. Linear quantifier elimination. *J. Automated Reasoning*, 45:189–212, 2010.

[9] Mojzesz Presburger. Uber die vollstandigkeiteines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt. In *Comptes-Rendus du ler Congres des Mathematiciens des Pays Slavs*, 1929.

[10] Alfred Tarski. A decision method for elementary algebra and geometry. 1951.

A List of Test Cases

```
ex01 : ∀ x : int, x ≤ -x → x ≤ 0
ex02 : ∀ x y : int, (x ≤ 5 ∧ y ≤ 3) → x + y ≤ 8
ex03 : ∀ x y z : int, x < y → y < z → x < z
ex04 : ∀ x y z : int, x - y ≤ x - z → z ≤ y
ex05 : ∀ x : int, (x = 5 ∨ x = 7) → 2 < x
ex06 : ∀ x : int, (x = -5 ∨ x = 7) → ¬x = 0
ex07 : ∀ x : int, 31 * x > 0 → x > 0
ex08 : ∀ x y : int, (-x - y < x - y) →
         (x - y < x + y) → (x > 0 ∧ y > 0)
```

```
ex09 : ∀ x : int, (x ≥ -1 ∧ x ≤ 1) →
       (x = -1 ∨ x = 0 ∨ x = 1)
ex10 : ∀ x : int, ∃ y : int, x = 2 * y ∨ x = (2 * y) + 1
ex11 : ∀ x : int, 5 * x = 5 → x = 1
ex12 : ∀ x y : int, ¬(2 * x + 1 = 2 * y)
ex13 : ∀ x y z : int, (2 * x + 1 = 2 * y) → x + y + z > 129
ex14 : ∃ x y : int, 5 * x + 3 * y = 1
ex15 : ∀ x y : int, x + 2 < y →
       ∃ z w : int, (x < z ∧ z < w ∧ w < y)
ex16 : ∀ x : int, (∃ y : int, 2 * y + 1 = x) →
       ¬∃ y : int, 4 * y = x
ex17 : ∀ x y z : int, x = y → y = z → x = z
ex18 : ∀ x : int, x < 349 ∨ x > 123
ex19 : ∀ x y : int, x ≤ 3 * y → 3 * x ≤ 9 * y
ex20 : ∃ x : int, 5 * x = 1335
ex21 : ∃ x y : int, x + y = 231 ∧ x - y = -487
ex22 : ∃ x y : int, 32 * x = 2023 + y
ex23 : ∀ x : int, (x < 43 ∧ x > 513) → ¬x = x
ex24 : ∀ x : int, ∃ y : int, x = 3 * y - 1 ∨
       x = 3 * y ∨ x = 3 * y + 1
ex25 : ∀ a : int, ∃ b : int, a < 4 * b + 3 * a ∨
       (¬(a < b) ∧ a > b + 1)
ex26 : ∃ x y : int, x > 0 ∧ y ≥ 0 ∧ 3 * x - 5 * y = 1
ex27 : ∃ x y : int, x ≥ 0 ∧ y ≥ 0 ∧ 5 * x - 6 * y = 1
ex28 : ∃ x y : int, x ≥ 0 ∧ y ≥ 0 ∧ 5 * x - 3 * y = 1
ex29 : ∃ x y : int, x ≥ 0 ∧ y ≥ 0 ∧ 3 * x - 5 * y = 1
ex30 : ∃ a b : int, ¬(a = 1) ∧ ((2 * b = a) ∨
       (2 * b = 3 * a + 1)) ∧ (a = b)
ex31 : ∀ x : int, ∃ y : int, x = 5 * y - 2 ∨ x = 5 * y - 1 ∨
       x = 5 * y ∨ x = 5 * y + 1 ∨ x = 5 * y + 2
ex32 : ∀ x y : int, 6 * x = 5 * y → ∃ d : int, y = 3 * d
ex33 : ∀ x : int, ¬(∃ m : int, x = 2 * m) ∧
       (∃ m : int, x = 3 * m + 1) ↔
       (∃ m : int, x = 12 * m + 1) ∨
       (∃ m : int, x = 12 * m + 7)
ex34 : ∀ x y : int, (∃ d : int, x + y = 2 * d) ↔
       ((∃ d : int, x = 2 * d) ↔ (∃ d : int, y = 2 * d))
ex35 : ∀ x : int, x > 5000 →
```

```
       ∃ y : int, y ≥ 1000 ∧ 5 * y < x
ex36 : ∀ x : int, (∃ y : int, 3 * y = x) →
       (∃ y : int, 7 * y = x) → (∃ y : int, 21 * y = x)
ex37 : ∀ y : int, (∃ d : int, y = 65 * d) →
       (∃ d : int, y = 5 * d)
ex38 : ∀ n : int, 0 < n ∧ n < 2400 →
       n ≤ 2 ∧ 2 ≤ 2 * n ∨
       n ≤ 3 ∧ 3 ≤ 2 * n ∨
       n ≤ 5 ∧ 5 ≤ 2 * n ∨
       n ≤ 7 ∧ 7 ≤ 2 * n ∨
       n ≤ 13 ∧ 13 ≤ 2 * n ∨
       n ≤ 23 ∧ 23 ≤ 2 * n ∨
       n ≤ 43 ∧ 43 ≤ 2 * n ∨
       n ≤ 83 ∧ 83 ≤ 2 * n ∨
       n ≤ 163 ∧ 163 ≤ 2 * n ∨
       n ≤ 317 ∧ 317 ≤ 2 * n ∨
       n ≤ 631 ∧ 631 ≤ 2 * n ∨
       n ≤ 1259 ∧ 1259 ≤ 2 * n ∨
       n ≤ 2503 ∧ 2503 ≤ 2 * n
ex39 : ∀ z : int, z > 7 → ∃ x y : int,
       x ≥ 0 ∧ y ≥ 0 ∧ 3 * x + 5 * y = z
ex40 : ∃ w x y z : int, 2 * w + 3 * x + 4 * y + 5 * z = 1
ex41 : ∀ x : int, x ≥ 8 → ∃ u v : int,
       u ≥ 0 ∧ v ≥ 0 ∧ x = 3 * u + 5 * v
ex42 : ∀ x y : int, x ≤ y → 2 * x + 1 < 2 * y
ex43 : ∀ a b : int, ∃ x : int, a < 20 * x ∧ 20 * x < b
ex44 : ∃ y : int, ∀ x : int, x + 5 * y > 1 ∧
       13 * x - y > 1 ∧ x + 2 < 0
ex45 : ∃ x y : int, 5 * x + 10 * y = 1
ex46 : ∀ x y : int, x ≥ 0 ∧ y ≥ 0 →
       12 * x - 8 * y < 0 ∨ 12 * x - 8 * y > 2
ex47 : ∃ x y : int, x ≥ 0 ∧ y ≥ 0 ∧ 6 * x - 3 * y = 1
ex48 : ∀ x y : int, ¬(x = 0) →
       5 * y < 6 * x ∨ 5 * y > 6 * x
ex49 : ∀ x y : int, ¬(6 * x = 5 * y)
ex50 : ∃ a b : int, a > 1 ∧ b > 1 ∧ ((2 * b = a) ∨
       (2 * b = 3 * a + 1)) ∧ (a = b)
```

Examples 1-41 are theorems, (roughly) in the order of increasing difficulty. 1-15 are easy theorems that can be proven by any version of `lia`, including the non-optimized prototype (now deprecated). 16-34 are medium difficulty theorems solved by the current version of `lia` under standard settings. 35-41 are hard theorems included for tests under increased time settings. 42-50 are nontheorems used for testing the soundness of `lia_vm`. Examples 1-11, 16-24, and 35 are my own; the rest are from Harrison [6].