Proof Synthesis and Reflection for Linear Arithmetic

Amine Chaieb and Tobias Nipkow

Institut für Informatik Technische Universität München
{chaieb,nipkow}@in.tum.de

Abstract. This article presents detailed implementations of quantifier elimination for both integer and real linear arithmetic for theorem provers. The underlying algorithms are those by Cooper (for Z) and by Ferrante and Rackoff (for $\mathbb{R}$). Both algorithms are realized in two entirely different ways: once in *tactic* style, i.e. by a proof-producing functional program, and once by *reflection*, i.e. by computations inside the logic rather than in the meta-language. Both formalizations are generic because they make only minimal assumptions w.r.t. the underlying logical system and theorem prover. An implementation in Isabelle/HOL shows that the reflective approach is between one and two orders of magnitude faster.

1 Introduction

Decision procedures have become an integral part of most automatic and interactive theorem provers. Their implementations come in three flavours: those that just say Yes or No, those that accompany their answer with a proof in some logical system, and those that have been defined and verified in a theorem prover. This paper is concerned with decision procedures of the latter two flavours for linear arithmetic over $\mathbb{Z}$ and $\mathbb{R}$. In our discussion we focus on proof-producing procedures first.

The obsession with proofs comes from the simple fact that complicated decision procedures, just like any tricky piece of software, are error prone. This can be a bit of an embarrassment in systems that are meant to raise the notion of logical correctness to a new level. Hence many theorem provers either produce proofs of some form or another or are based on the LCF paradigm, where theorems are an abstract data type whose interface are the inference rules of the logic.

The implementation of proof-producing decision procedures is very subtle because one needs to ensure that when theorems are plugged together they really fit together. For example, combining $0 < x + 0$ and $0 < x \to 0 < 2{\cdot}x$ by modus ponens will not produce the theorem $0 < 2{\cdot}x$ but will fail, since futher inference is needed to prove that $0 < x + 0$ and

$0 < x$ are equivalent. Such bugs do not endanger soundness but are still quite annoying. They may lurk in the code for a long time because they are not caught by a standard static type system which cannot express the precise form the theorem produced or expected by some function must have (but see [32]). This problem is exacerbated by the fact that decision procedures are re-implemented time and again for different systems, and that in the literature these implementations are only sketched if discussed at all because it is not considered scientific enough. If we want to make progress we need to increase the amount of sharing in this area drastically. Unfortunately the actual implementation level is unsuitable for that because it is too intertwined with the specifics of the theorem prover a decision procedure is written for.

Hence we argue that the field is in need of descriptions of proof-producing decision procedures at a level that is abstract enough to be sharable across different theorem provers yet close enough to the implementation level to be readily implemented. In this paper we give two examples of such detailed yet generic descriptions: proof-producing quantifier elimination procedures for linear arithmetic over $\mathbb{Z}$ and $\mathbb{R}$. These theories are ubiquitous and tricky to implement. We hope that over time a library of such decision procedures will be published. This should be one way of bridging the gap between the many different theorem provers currently in use, at least by reducing the amount of work that is duplicated time and again.

Our description of the decision procedures tries to be as abstract as possible by using some generic functional programming language and assuming only a minimal set of capabilities of the underlying theorem prover. Neither are we very specific about the underlying logic, except that we assume it is classical. As evidence of the genericity of our code we can offer that the procedure for Presburger arithmetic is derived from an implementation for Isabelle/HOL [40] by the first author [8], yet is quite close to an implementation in the HOL system [25].

Producing proofs requires no meta-theory but has many disadvantages: (a) the actual implementation requires intimate knowledge of the internals of the underlying theorem prover; (b) there is no way to check at compile type if the proofs will really compose; (c) it is inefficient because one has to go through the inference rules in the kernel; (d) if the prover is based on proof objects this can lead to excessive space consumption (proofs may require (super-) exponential space [20, 43]).

These shortcomings can be overcome by defining and verifying the decision procedure inside the logic, provided one can perform the computations thus expressed in an efficient manner, typically by compiling them into some programming language, executing them there, and importing the result back into the logic. This is often called *reflection*. The drawbacks are that it requires a logic that contains an executable subset, it requires the user to prove the correctness of the decision procedure, and it does not yield a proof term (which may or may not be desirable). Hence both approaches are common implementation techniques for decision procedures. The technical contributions of the paper are the application of both techniques to quantifier elimination over $\mathbb{Z}$ and $\mathbb{R}$. We provide

- the first exposition of a generic proof-producing implementation of Cooper's algorithm;
- the first completely reflective implementation of Cooper's algorithm;
- the first proof-producing and reflective implementations of Ferrante and Rackoff's algorithm.

It should be emphasized that although we are fully aware of the computational complexity of especially Cooper's algorithm, we have refrained from optimizing our implementations. They are meant as easy to understand starting points for other people's implementations, not as highly competitive pieces of code.

The paper is structured as follows. After an introduction to the two quantifier elimination algorithms (§2) they are first formalized as proof-producing functions in the meta-language (§3) and then formalized and verified in logic (§4).

Related work

Fourier [21] presented an extension of Gauss-elimination to cope with inequalities. His method has been rediscovered several times [17, 37] and is today referred to as *Fourier-Motzkin elimination*. Tarski's result for real closed fields [50] also yields a decision procedure for linear real arithmetic. More efficient quantifier elimination procedures have been developed by Ferrante and Rackoff [19], the work on which we rely, and later by Weispfenning [51] and Loos and Weispfenning [33]. Quantifier elimination procedures for Presburger Arithmetic have been discovered by Presburger [45] and Skolem [49] independently and improved by Cooper [13], Reddy and Loveland [47] and extended to deal with parameters by Weispfenning [52]. Pugh [46] presented an adaptation of Fourier-Motzkin elimination to cope with integers. An automata based method is presented in [55].

Linear arithmetic enjoys exciting complexity results. Fischer and Rabin [20] proved lower bounds which were later refined [3, 4, 23] using alternation [11]. Upper bounds can be found in [43, 19, 51, 33]. The complexity of subclasses of Presburger arithmetic is also well studied [48, 26]. Weispfenning [53, 52, 51, 33] provides precise bounds for the quantifier elimination problem allowing (non linear) parameters. A bound on the automata size is due to Klaedtke [31].

The first implementation of a decision procedure for Presburger arithmetic, and in fact the first theorem prover, dates back to 1957 [16]. But in the following we mostly focus on those papers that are concerned with proof-producing decision procedures.

Norrish [41] discusses proof-producing implementations in HOL [25] of Cooper's algorithm (in tactic-style) and Pugh's algorithm (by reflection of a proof trace found on the meta-level). The key difference to our paper is that he discusses design principles on an abstract level but omits the details of proof synthesis. For that he refers to the actual code, which we would argue is much too system specific to be easily portable. Crégut [14] presents an implementation of Pugh's method for Coq [5], using the same technique as Norrish. His implementation only deals with quantifier-free Presburger arithmetic and is even there incomplete. For the reals most theorem provers implement Fourier-Motzkin elimination and only deal with quantifier-free formulae. HOL Light [28] includes full quantifier-elimination for real closed fields [30, 36]. Verifying the CAD algorithm [12] is on-going work [34]. An alternative approach already used in [38] is based on checking certificates for Farkas's lemma. This technique is very efficient, but is not complete for the integers and does not allow quantifier elimination, which is the main focus of this paper.

The method of reflection goes back at least to the meta-functions used by Boyer and Moore [7] and later became popular in theorem provers based on type theory [5]. It has been studied by several researchers [29, 1]. Our use of reflection is rather computational and has nothing to do with "logical reflection" [29]. Laurent Théry verified Presburger's original algorithm in Coq (see the Coq home page). We also verified Cooper's algorithm in Isabelle [10].

Finally note that the first-order theory of linear arithmetic over both reals and integers also admits quantifier elimination [54]. We verified this algorithm in Isabelle [9]. An automata based algorithm [6] has also been presented to solve the decision problem (not the quantifier elimination problem).

2 Quantifier elimination for linear arithmetic

This section provides an informal introduction to linear arithmetic over $\mathbb{Z}$ and $\mathbb{R}$ and to their quantifier elimination procedures due to Cooper [13] and Ferrante and Rackoff [19] respectively.

2.1 Linear arithmetic

We consider $\mathcal{Z}_+$ and $\mathcal{R}_+$, the first-order theories of linear arithmetic over $\mathbb{Z}$ and $\mathbb{R}$ respectively. The formulae are defined by means of first order logic over the atoms $s = t$ and $s < t$, where s and t are terms built up from variables $(x, y, z \dots)$, 0, 1 and addition $+$. To permit quantifier elimination $\mathcal{Z}_+$ also includes atoms of the form $d \mid t$, where $d \in \mathbb{Z}$, which expresses that d divides t. We also use $d \nmid t$ as a shorthand for $\neg(d \mid t)$. We allow the use of constants $c \in \mathbb{Z}$ for $\mathcal{Z}_+$ and $c \in \mathbb{Q}$ for $\mathcal{R}_+$ respectively, and allow multiplication by these constants.

2.2 Quantifier elimination and normalized formulae

Both $\mathcal{Z}_+$ and $\mathcal{R}_+$ admit quantifier elimination [45, 49, 47, 19, 51, 33]. Note that if *qe* is a method for eliminating $\exists$ from $\exists x.P(x)$, where $P(x)$ is quantifier-free, then applying *qe* recursively to the innermost quantifiers first yields a quantifier elimination procedure for the whole theory. In fact $P(x)$ has only to be in a representative syntactical subset of quantifier-free formulae. Hence in the following we only describe how to eliminate one $\exists$ from $\exists x.P(x)$, for a normalized formula $P(x)$.

Normalized formulae

A quantifier-free formula $P(x)$ is *x-normalized* if it is built up from $\wedge, \vee$ and atomic formulae of type (A) $x = t$, (B) $x < t$, (C) $t < x$ or (N) those not depending on x. For $\mathcal{Z}_+$ the atoms (D) $d \mid x + t$ and (E) $d \nmid x + t$ are also allowed. The term t, in (A)-(E), does not involve x.

In the case of $\mathcal{R}_+$ it is easy to see that any quantifier-free formula $P(x)$ can be transformed into x-normalized form by means of negation normal form (NNF), elimination of negations and multiplication by appropriate rational numbers. For $\mathcal{Z}_+$ this is performed as follows:

1. Put $P(x)$ in NNF and transform $\neg(s < t)$ into $t < s + 1$ and $\neg(s = t)$ into $t < s \vee s < t$.

2. Transform every atom according to the coefficient of x into (A') $c{\cdot}x = t$, (B') $c{\cdot}x < t$, (C') $t < c{\cdot}x$, (D') $d \mid c{\cdot}x + t$, (E') $d \nmid c{\cdot}x + t$ or (N), where $c > 0$, $d > 0$ and x does not occur in t. Note that $d \mid t = -d \mid t$ and $d \mid t = d \mid -t$ hold in $\mathcal{Z}_+$.
3. Compute $l = lcm\{c \mid$ '$c{\cdot}x$' occurs in an atom$\}$ and multiply every atom containing $c{\cdot}x$ by $\frac{l}{c}$. Note that the resulting formula $Q(x)$ is equivalent to $P(x)$ and that the coefficient of x is now l everywhere. Hence we can view $Q(x)$ as $Q'(l{\cdot}x)$ for an appropriate Q'.
4. Return $l \mid x \wedge Q'(x)$ according to the generic theorem

$$\textit{unitcoeff}: \quad (\exists x.\ Q(l{\cdot}x)) \leftrightarrow (\exists x.\ l \mid x \wedge Q(x))$$

See also [18] for good examples of normalization.

We restrict the atomic relations to $=$ and $>$ only for the sake of presentation. In practice it is important though to include $\leq$ and $\neq$, which prevent a fatal blow up in normalization such as the reduction of $s \neq t$ to $s > t \vee t < s$ above. It is straightforward to extend the algorithms in §2.3 and §2.4 to deal with $\leq$ and $\neq$ in addition to $<$ and $=$.

Elimination sets

An important technical device shared by both algorithms are *elimination sets* [33]. Given an x-normalized formula $P(x)$, Cooper's algorithm computes $\mathcal{B}_P$ and $P_{-\infty}$ (alternatively $\mathcal{A}_P$ and $P_{+\infty}$) and Ferrante and Rackoff's algorithm computes $\mathcal{U}_P$, $P_{-\infty}$ and $P_{+\infty}$ as defined in Fig. 1. Note that the table in Fig. 1 is a compact representation of a set of recursive equations. In more conventional notation we would have written $\mathcal{U}(F \wedge G) = \mathcal{U}(F) \cup \mathcal{U}(G)$ for the entry under $\mathcal{U}_P$ and $F \wedge G$. Note that the definitions are recursive and implicitly depend on the bound variable x: the final line applies in case the sub-formula under consideration does not match any of the previous lines.

Example 1. Consider the formula P (implicitly depending on the bound variable x) $x < t \wedge x > 2 \vee x < t + y \wedge y < 1$. Then $\mathcal{U}_P = \{t, 2, t + y\}$ is the set of all lower and upper bounds of x (considered as a real number). Analogously $\mathcal{A}_P = \{\ t, t{+}y\}$ is the set of all upper bounds of x (considered as an integer), and $\mathcal{B}_P = \{2\}$ is the set of all lower bounds of x (considered as an integer). The formulae $P_{-\infty} = \textit{True} \wedge \textit{False} \vee \textit{True} \wedge y < 1$ and $P_{+\infty} = \textit{False} \wedge \textit{True} \vee \textit{False} \wedge y < 1$ represent the substitution of very large negative and positive numbers, respectively, for x in P.

Cooper's algorithm inspired Ferrante and Rackoff [19], which explains the similiarities of the computed sets.

P	$\mathcal{U}_P$	$\mathcal{B}_P$	$\mathcal{A}_P$	$P_{-\infty}$	$P_{+\infty}$
$F \wedge G$	$\mathcal{U}_F \cup \mathcal{U}_G$	$\mathcal{B}_F \cup \mathcal{B}_G$	$\mathcal{A}_F \cup \mathcal{A}_G$	$F_{-\infty} \wedge G_{-\infty}$	$F_{+\infty} \wedge G_{+\infty}$
$F \vee G$	$\mathcal{U}_F \cup \mathcal{U}_G$	$\mathcal{B}_F \cup \mathcal{B}_G$	$\mathcal{A}_F \cup \mathcal{A}_G$	$F_{-\infty} \vee G_{-\infty}$	$F_{+\infty} \vee G_{+\infty}$
$t < x$	$\{t\}$	$\{t\}$	$\emptyset$	*False*	*True*
$x < t$	$\{t\}$	$\emptyset$	$\{t\}$	*True*	*False*
$x = t$	$\{t\}$	$\{t-1\}$	$\{t+1\}$	*False*	*False*
_	$\emptyset$	$\emptyset$	$\emptyset$	P	P

Fig. 1. Definition of $\mathcal{U}_P$, $\mathcal{B}_P$, $\mathcal{A}_P$, $P_{-\infty}$ and $P_{+\infty}$

The underlying idea is to compute a finite set $\mathcal{S}$ of terms such that $\exists x. P(x)$ is equivalent to $\bigvee_{t \in \mathcal{S}} P(t)$. The terms in $\mathcal{S}$ are sometimes called Skolem terms [51, 33] because they are the concrete witnesses as opposed to abstract Skolem functions. The formula $P_{+\infty}$ represents the result of substituting "large" potential witnesses for x in an x-normalized formula $P(x)$.

This completes the exposition of the common basis of the two algorithms. Now we study their particularities. In both presentations, we include proofs for the main theorems. These already outline the general strategies for the proof-procedures and are important for subsequent details.

2.3 Cooper's algorithm

The input to Cooper's algorithm is a formula $\exists x. P(x)$, where $P(x)$ is an x-normalized $\mathcal{Z}_+$-formula. Besides $\mathcal{A}_P$, $\mathcal{B}_P$, $P_{-\infty}$ and $P_{+\infty}$, cf. Fig. 1, the algorithm computes

$$\delta = lcm\{d \mid \text{`}d \mid x + t\text{' occurs in } P\}. \tag{1}$$

The result is obtained by applying either (2) or (3) in Cooper's Theorem:

Theorem 1 (Cooper [13]) *If $P(x)$ is an x-normalized $\mathcal{Z}_+$ formula then*

$$\exists x. P(x) \;\leftrightarrow\; \exists j \in [\delta]. P_{-\infty}(j) \;\vee \exists j \in [\delta], b \in \mathcal{B}_P. P(b + j) \tag{2}$$

$$\exists x. P(x) \;\leftrightarrow\; \exists j \in [\delta]. P_{+\infty}(j) \;\vee \exists j \in [\delta], b \in \mathcal{A}_P. P(a - j) \tag{3}$$

The right-hand side is quantifier-free, since $\exists$ ranges over finite sets: $\mathcal{B}_P$ and $[\delta] = \{1..\delta\}$. The choice of which of the two equivalences to apply, (2) or (3), is normally determined by the relative size of $\mathcal{A}_P$ and $\mathcal{B}_P$.

Proof. We give a simpler version of the proof by Norrish [41]. We only show the proof of (2). The proof of (3) is analogous.

Obviously, if $\exists j \in [\delta], b \in \mathcal{B}_P. P(b + j)$ holds then $\exists x. P(x)$ trivially holds. Hence to finish the proof we only need to prove $\exists j \in [\delta]. P_{-\infty}(j) \rightarrow \exists x. P(x)$ and $(\exists x. P(x)) \wedge \neg(\exists j \in [\delta], b \in \mathcal{B}_P. P(b + j)) \rightarrow \exists j \in [\delta]. P_{-\infty}(j)$.

1. $\exists j \in [\delta].P_{-\infty}(j) \longrightarrow \exists x.P(x)$ To prove this, we need the following properties of $P_{-\infty}$:

$$\exists z.\forall x < z.P(x) \leftrightarrow P_{-\infty}(x) \tag{4}$$

$$\forall x, k.P_{-\infty}(x) \leftrightarrow P_{-\infty}(x - k{\cdot}\delta). \tag{5}$$

These properties are proved by induction on P: (4) states that $P(x)$ and $P_{-\infty}(x)$ coincide over arguments that are small enough ; (5) states that $P_{-\infty}(x)$ is unaffected by the substraction of any number of multiples of δ, i.e. $\{x \mid P_{-\infty}(x)\}$ is a periodic set. Note that only divisibility relations $d \mid x + r$ occur in $P_{-\infty}$, where $d \mid \delta$ (cf. (1)).

Now assume that $P_{-\infty}(j)$ holds for some $1 \leq j \leq \delta$, then using (5) we can substract enough multiples of δ to reach a number below the z from (4), and thus obtain by (4) a witness for P.

2. $(\exists x.P(x)) \wedge \neg(\exists j \in [\delta], b \in \mathcal{B}_P.P(b + j)) \longrightarrow \exists j \in [\delta].P_{-\infty}(j)$ Again due to (5) and (4), i.e. the argument above of decreasing witnesses by multiples of δ, it is sufficient to prove

$$\forall x.\neg(\exists j \in [\delta], b \in \mathcal{B}_P.x = b + j) \rightarrow P(x) \rightarrow P(x - \delta). \tag{6}$$

The proof of (6) is by induction on P. The cases $\wedge$ and $\vee$ are trivial. In the case $x = t$ we derive a contradiction by taking $j = 1$, since $t - 1 \in \mathcal{B}_P$. In the cases $x < t$ and $d \mid x + t$ the claim is immediate since since $\delta > 0$ and $d \mid \delta$, respectively. For the case $t < x$, assume that $t + \delta \geq x$. Hence $x = t + j$ for some $1 \leq j \leq \delta$, which contradicts the assumption since $t \in \mathcal{B}_P$.

2.4 Ferrante and Rackoff's algorithm

The input to Ferrante and Rackoff's algorithm is a formula $\exists x.P(x)$, where $P(x)$ is an x-normalized $\mathcal{R}_+$-formula. The algorithm just applies Ferrante and Rackoff's theorem.

Theorem 2 (Ferrante and Rackoff [19]) *If $P(x)$ is an x-normalized $\mathcal{R}_+$-formula then*

$$\exists x.P(x) \leftrightarrow P_{-\infty} \vee P_{+\infty} \vee \exists (u, u') \in \mathcal{U}_P^2.P(\frac{u + u'}{2}).$$

Proof. Although the proof in [19] is mathematically clear we give a more formal proof, which is suitable in a theorem prover setting.

First we show the "←"-direction. Obviously $\exists(u,u') \in \mathcal{U}_P^2.P(\frac{u+u'}{2}) \to \exists x.P(x)$ holds. The cases $P_{-\infty} \to \exists x.P(x)$ and $P_{+\infty} \to \exists x.P(x)$ follow directly from (4) and and its dual (7) for $P_{+\infty}$, i.e. the fact that $P_{+\infty}$ simulates the behavior of P for arbitrarily large positive real numbers.

$$\exists y.\forall x > y.P(x) \leftrightarrow P_{+\infty} \tag{7}$$

For the "→"-direction assume $P(x)$ for some x and $\neg P_{-\infty}$ and $\neg P_{+\infty}$, i.e. x is neither "too large" nor "too small" a witness for P. We first prove that x must lie between two points in $\mathcal{U}_P$, a trivial consequence of (8) and (9), both proved by induction.

$$\forall x.\neg P_{-\infty} \wedge P(x) \to \exists l \in \mathcal{U}_P.l \leq x \tag{8}$$
$$\forall x.\neg P_{+\infty} \wedge P(x) \to \exists u \in \mathcal{U}_P.x \leq u \tag{9}$$

Now we conclude that either $x \in \mathcal{U}_P$, in which case we are done since $\frac{x+x}{2} = x$, or we can find the smallest interval with endpoints in $\mathcal{U}_P$ containing x, i.e. $l_x < x < u_x \wedge \forall y.l_x < y < u_x \to y \notin \mathcal{U}_P$ for some $(l_x, u_x) \in \mathcal{U}_P^2$. The construction of this smallest interval is simple since $\mathcal{U}_P$ is finite and reals are totally ordered.

Now we prove $P(y)$ for all $y \in (l_x, u_x)$, and hence finish the proof by taking $u = l_x$ and $u' = u_x$. This property shows the expressive limitations of $\mathcal{R}_+$: P does *not* change its truth value over smallest intervals with endpoints in $\mathcal{U}_P$, i.e.

$$\begin{aligned}\forall x, l, u.(\forall y.l < y < u \to y \notin \mathcal{U}_P) \wedge l < x < u \wedge P(x) \\ \to \forall y.l < y < u \to P(y)\end{aligned} \tag{10}$$

The proof of (10) is by induction on P. If P is of type (A) the result holds because the premise is contradictory. For the case $x < t$, fix an arbitrary y and assume $l < y < u$. Note that $y \neq t$ since $t \in \mathcal{U}_P$. Hence $y < t$, i.e. $P(y)$, for if $y > t$ then $l < t < u$, which contradicts the premises since $t \in \mathcal{U}_P$. The $x > t$ case is analogous and the $\wedge$ and $\vee$-cases are trivial.

3 Proof synthesis

This section introduces an abstract generic framework for describing decision procedures as functional programs in a meta-language for manipulating theorems. Then the algorithms by Cooper and by Ferrante and Rackoff are described in this meta-language.

3.1 Notation

Logic Terms and formulae follow the usual syntax of predicate calculus. However, there are two levels that we must distinguish. On the programming language level, i.e. the implementation level, the type of formulae is an ordinary first-order recursive datatype. In particular, the formula $\exists x.A$ can be decomposed (e.g. by pattern matching) into the bound variable x and the formula A, which may contain x. On the logic level, we assume that our language allows predicate variables, as is the case in all higher-order systems. On this level $\exists x.A$ is a formula where A does not depend on x, whereas P in $\exists x.P(x)$ is a predicate variable, i.e. a *function* from terms to formulae, which is applied to x, thus expressing the dependence on x. One advantage of the higher-order notation is that substituting x by t is expressed by moving from $P(x)$ to $P(t)$.

Theorems can only be proved with a fixed set of functions which we discuss now. If there is no danger of confusion we identify a theorem with the formula it proves. But in order not to blur the distinction too much, we usually display theorems in an *inference rule* style

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A \tag{11}$$

where the A_i are the premises and A is the conclusion. Logically this is equivalent to $A_1 \to \cdots \to A_n \to A$ but it emphasizes that it is a theorem.

We will now discuss the interface of the theorem data type which offers the following functions: a generalized form of modus ponens (*fwd*), instantiation of free variables, generalization (*gen*), and some basic arithmetic capabilities.

If th is (11), and $th_1, \ldots, th_n$ are theorems that match the premises $A_1, \ldots, A_n$, then $\mathit{fwd}\, th\ [th_1, \ldots, th_n]$ produces the corresponding instance of A. That is, if $B_1, \ldots, B_n$ are the formulae proved by the theorems th_1, $\ldots$, th_n, and if θ is a matching substitution for the set of equations $A_1 = B_1, \ldots, A_n = B_n$, i.e. $\theta(A_i) = B_i$, then $\mathit{fwd}\ th\ [th_1, \ldots, th_n]$ yields the theorem $\theta(A)$.

The *free* variables in a theorem th can be instantiated from left to right with terms $t_1, \ldots, t_n$ by writing $th[t_1, \ldots, t_n]$. For example, if th is the theorem $m \le m + n{\cdot}n$ then $th[1, 2]$ is the theorem $1 \le 1 + 2{\cdot}2$.

Function *gen* performs ∀-introduction: it takes a variable x and a theorem $P(x)$ and returns the theorem $\forall x.P(x)$.

Note that we assume that *fwd* performs higher-order matching, but only of a very simple kind. Its first argument th may be a theorem containing a predicate variable P. In this case there must be one occurrence of $P(x)$ among the premises of th such that x is a bound variable. This guarantees that there is at most one matching substitution because it is a special case of pattern-unification [39]. Further occurrences of P among the premises cannot lead to further matching substitutions but can only rule some out. Hence it is justified to speak of *the* matcher. Furthermore this kind of higher-order matching is straightforward to implement.

Programming language All algorithms are expressed in generic functional programming notation. We assume the following special features. Lambda-abstraction uses a bold $\boldsymbol{\lambda}$ (in contrast to the ordinary λ on the logic level) and permits pattern-matching (where most uses are of the form $\boldsymbol{\lambda}[].t$, where $[]$ is the empty list). Because formulae (a concrete recursive type) and theorems (some abstract type) are quite distinct, we need a way to refer to the formula proved by some theorem. This is done by pattern matching: a theorem can be matched against the pattern th **as** 'f', where th is a theorem variable and f a formula pattern, thus binding the formula variables in f. For example, matching the theorem $0 = 0 \wedge 1 = 1$ against the pattern th **as** '$A \wedge B$' binds th to the given theorem, A to the term $0 = 0$ and B to the term $1 = 1$.

Patterns may be guarded by boolean conditions as in Haskell: $p \mid b_1, b_2$ is the pattern p that is guarded by the conditions b_1 and b_2.

3.2 The theorem extraction model

Many of our proofs are performed by the following generic function:

```
thm decomp t =
let
  (ts,recomb) = decomp t
in recomb (map (thm decomp) ts)
```

It takes a problem decomposition function of type $\alpha \to \alpha\ list \times (\beta\ list \to \beta)$ and a problem t of type α, decomposes t into a list of subproblems ts and a recombination function *recomb*, solves the subproblems recursively, and combines their solution into an overall solution.

In our applications, problems are formulae to be proved, solutions are theorems, and termination will be guaranteed because all decompositions yield smaller terms. This style of theorem proving was invented with the LCF system [24, 44], where *decomp* is called a *tactic*. Hence we refer to it as *tactic-style theorem proving*. Note that the interface of our theorem data type is quite abstract. Theorems may be implemented as full-blown proof terms, where the whole derivation is stored, or as in LCF, where only the proved formula is retained.

Example 2. As a first example we present a generic function *qelim* which eliminates all quantifiers from a first-order formula provided it is given a function *qe* which can eliminate a single existential quantifier. That is, if *qe* applied to a formula $\exists x.P$, where P is quantifier free, yields a theorem $(\exists x.P(x)) \leftrightarrow Q$, where Q is quantifier free, then *qelim qe* applied to any first-order formula F yields a theorem $F \leftrightarrow F'$, where F' is quantifier free. Elimination proceeds from the innermost quantifier to the outermost one. The function is theory independent. It uses the following predicate calculus tautologies:

$\mathit{cong}_{\Diamond} : [\![P \leftrightarrow P'; Q \leftrightarrow Q']\!] \Longrightarrow P \Diamond Q \leftrightarrow P' \Diamond Q'$, for $\Diamond \in \{\wedge, \vee, \rightarrow, \leftrightarrow$
$\}$
$\mathit{cong}_{\neg} : [\![P \leftrightarrow P']\!] \Longrightarrow \neg P \leftrightarrow \neg P'$
$\mathit{cong}_{\exists} : [\![\forall x.P(x) \leftrightarrow Q(x)]\!] \Longrightarrow \exists x.P(x) \leftrightarrow \exists x.Q(x)$
$\mathit{qe}_{\forall} : [\![(\exists x.\neg P(x)) \leftrightarrow R]\!] \Longrightarrow (\forall x.P(x)) \leftrightarrow \neg R$
$\mathit{trans}\colon [\![P \leftrightarrow Q; Q \leftrightarrow R]\!] \Longrightarrow P \leftrightarrow R$
$\mathit{refl} : P \leftrightarrow P$

The actual function definition is straightforward:

```
$decomp\_qe\ qe\ P =$
case $P$ of
  $F \Diamond G \mid \Diamond \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \Rightarrow ([F, G], \mathit{fwd}\ \mathit{cong}_{\Diamond})$
  $\neg F \Rightarrow ([F], \mathit{fwd}\ \mathit{cong}_{\neg})$
  $\exists x.F \Rightarrow ([F], \boldsymbol{\lambda}[th$ as '$\_ \leftrightarrow G$'$]$. let $\mathit{lift} = \mathit{fwd}\ \mathit{cong}_{\exists}\ [\mathit{gen}\ x\ th]$
                                  in $\mathit{fwd}\ \mathit{trans}\ [\mathit{lift}, \mathit{qe}\ x\ G])$
  $\forall x.F \Rightarrow ([\exists x.\neg F], \mathit{fwd}\ \mathit{qe}_{\forall})$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{fwd}\ \mathit{refl}[P])$

$qelim\ qe = thm\ (decomp\_qe\ qe)$
```

As an example, assume that we have a quantifier elimination function g to perform Gauß-elmination and consider the formula $P = (\exists z.3{\cdot}x = z \wedge 2{\cdot}z = 5)$. The call *qelim g P* becomes *thm* (*decomp_qe g*) *P*. By definition of *thm* we first compute *decomp_qe g P*. Because P starts with $\exists$, this yields $([F], f)$ where $F = (3{\cdot}x = z \wedge 2{\cdot}z = 5)$ and f is the recombination function given in the $\exists$-case of *decomp_qe*. By definition of *thm* this yields $f(\mathit{map}\ (\mathit{thm}(\mathit{decomp\_qe}\ g))\ [F])$. The expression *thm* (*decomp_qe g*) *F* performs elimination of all quantifiers inside F: F

is split in two subformulae $3{\cdot}x = z$ and $2{\cdot}z = 5$ which lead to the trivial theorems $3{\cdot}x = z \leftrightarrow 3{\cdot}x = z$ and $2{\cdot}z = 5 \leftrightarrow 2{\cdot}z = 5$ (via the last case in *decomp_qe*) which are then combined via $\mathit{cong}_\wedge$ into the theorem $F \leftrightarrow F$. Thus the call $f(\mathit{map}\ (\mathit{thm}(\mathit{decomp\_qe}\ g))\ [F])$ has become $f(F \leftrightarrow F)$. Now the f from the $\exists$-case in *decomp_qe* kicks in to eliminate the topmost quantifier $\exists z$. Because f is defined by pattern matching, G becomes F. First $F \leftrightarrow F$ is generalized to $\forall z.F \leftrightarrow F$ which $\mathit{cong}_\exists$ turns into $lift = (\exists z.F) \leftrightarrow (\exists z.F)$. The subterm $g\ z\ G$ calls Gauß-elmination and yields the theorem $(\exists z.F) \leftrightarrow 3{\cdot}x = \frac{5}{2}$. By transitivity with $lift$, the same theorem is produced and the evaluation of $f(F \leftrightarrow F)$ (and thus of $\mathit{qelim}\ g\ P$) terminates.

Example 3. As a second example we use *thm* to implement the normalization step, see §2.2.

$$\begin{aligned}
&\mathit{nnf}_{\rightarrow}\colon \llbracket \neg P \leftrightarrow P_1; Q \leftrightarrow Q_1 \rrbracket \Longrightarrow ((P \rightarrow Q) \leftrightarrow (P_1 \vee Q_1))\\
&\mathit{nnf}_{\leftrightarrow}\colon \llbracket (P \wedge Q) \leftrightarrow (P_1 \wedge Q_1); (\neg P \wedge \neg Q) \leftrightarrow (P_2 \wedge Q_2) \rrbracket\\
&\quad \Longrightarrow (P \leftrightarrow Q) \leftrightarrow (P_1 \wedge Q_1) \vee (P_2 \wedge Q_2)\\
&\mathit{nnf}_{\neg\neg}\colon \llbracket P \leftrightarrow Q \rrbracket \Longrightarrow \neg\neg P \leftrightarrow Q\\
&\mathit{nnf}_{\neg\wedge}\colon \llbracket \neg P \leftrightarrow P_1; \neg Q \leftrightarrow Q_1 \rrbracket \Longrightarrow \neg(P \wedge Q) \leftrightarrow (P_1 \vee Q_1)\\
&\mathit{nnf}_{\neg\vee}\colon \llbracket \neg P \leftrightarrow P_1; \neg Q \leftrightarrow Q_1 \rrbracket \Longrightarrow \neg(P \vee Q) \leftrightarrow (P_1 \wedge Q_1)\\
&\mathit{nnf}_{\neg\rightarrow}\colon \llbracket P \leftrightarrow P_1; \neg Q \leftrightarrow Q_1 \rrbracket \Longrightarrow \neg(P \rightarrow Q) \leftrightarrow (P_1 \wedge Q_1)\\
&\mathit{nnf}_{\neg\leftrightarrow}\colon \llbracket (P \wedge \neg Q) \leftrightarrow (P_1 \wedge Q_1); (\neg P \wedge Q) \leftrightarrow (P_2 \wedge Q_2) \rrbracket\\
&\quad \Longrightarrow \neg(P \leftrightarrow Q) \leftrightarrow ((P_1 \wedge Q_1) \vee (P_2 \wedge Q_2))
\end{aligned}$$

With these predicate calculus tautologies we obtain decomposition for NNF:

```
$\mathit{decomp\_nnf}\ \mathit{lf}\ P =$
**case** $P$ **of**
  $F \Diamond G \mid \Diamond \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \Rightarrow ([F, G],\ \mathit{fwd}\ \mathit{cong}_\Diamond)$
  $\neg\neg F \Rightarrow ([F],\ \mathit{fwd}\ \mathit{nnf}_{\neg\neg})$
  $\neg(F \wedge G) \Rightarrow ([\neg F, \neg G],\ \mathit{fwd}\ \mathit{nnf}_{\neg\wedge})$
  $\neg(F \vee G) \Rightarrow ([\neg F, \neg G],\ \mathit{fwd}\ \mathit{nnf}_{\neg\vee})$
  $\neg(F \rightarrow G) \Rightarrow ([F, \neg G],\ \mathit{fwd}\ \mathit{nnf}_{\neg\rightarrow})$
  $\neg(F \leftrightarrow G) \Rightarrow ([F \wedge \neg G, \neg F \wedge G],\ \mathit{fwd}\ \mathit{nnf}_{\neg\leftrightarrow})$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{lf}\ P)$
```

Literals are taken care of by parameter *lf* which, when applied to a literal L returns a theorem $L \leftrightarrow F$, where F should be in NNF. In our case we pass the function *proveL* as argument for *lf*. It takes a variable x and a literal L and returns the theorem $L \leftrightarrow F$ where F is in NNF and its atoms are of type (A)-(C) or (N) if x is a real variable and of type (A')-(E') or (N) if x is an integer variable (see page 5 for (A), (A') etc). This requires the manipulation of individual (in)equalities between linear terms. How this is best handled depends on the infrastructure of the underlying theorem prover: rewriting or quantifier-free linear arithmetic are possible implementation tools. Details are explained in the reflective approach. Hence we refrain from giving a generic solution for *proveL*. The normalization for $\mathcal{R}_+$ is hence simply defined by

$$\mathit{normalize}_{\mathcal{R}_+}\ x\ P = \mathit{thm}\ (\mathit{decomp\_nnf}\ (\mathit{proveL}\ x))\ P.$$

Normalization for $\mathcal{Z}_+$ is performed by $\mathit{normalize}_{\mathcal{Z}_+}$ and it proceeds as in §2.2: the first and second step are performed by *decomp_nnf* and *proveL*. The third step, adjusting the coefficient of x, is performed by *decomp_ac*:

$$\begin{aligned}&ac_{\bowtie}\text{: } [\![k > 0]\!] \Longrightarrow (c{\cdot}x \bowtie t) \leftrightarrow (k{\cdot}c{\cdot}x \bowtie k{\cdot}t),\ \text{for} \bowtie\, \in \{=, <, >\}\\ &ac_{\bowtie}\text{: } [\![k > 0]\!] \Longrightarrow (d \mid c{\cdot}x + t) \leftrightarrow (k{\cdot}d \mid k{\cdot}c{\cdot}x + k{\cdot}t),\ \text{for} \bowtie\, \in \{\mid, \nmid\}\end{aligned}$$

```
$\mathit{decomp\_ac}\ x\ \mathit{ldiv}_{>}\ P =$
$\mathbf{case}\ P\ \mathbf{of}$
  $F \Diamond G \mid \Diamond \in \{\wedge, \vee\} \Rightarrow ([F, G], \mathit{fwd}\ \mathit{cong}_{\Diamond})$
  $c{\cdot}y \bowtie t \mid y = x, \bowtie\{=, <, >\} \Rightarrow ([], (\mathit{fwd}\ ac_{\bowtie}\ [\mathit{ldiv}_{>}\ c])[c, x, t])$
  $d \bowtie c{\cdot}y + t \mid \bowtie\, \in \{\mid, \nmid\}, y = x \Rightarrow ([], (\mathit{fwd}\ ac_{\bowtie}\ [\mathit{ldiv}_{>}\ c])[d, c, x, t])$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\mathit{refl}[P])$
```

The argument $\mathit{ldiv}_{>}$ is a function that, given c, returns the theorem '$k > 0$', where k is numeral representing $\frac{l}{c}$ and $l = \mathit{lcm}\{c \mid \mathbf{as}\ \text{'}c{\cdot}x\text{'}\ \text{occurs in}\ P\}$. The final step takes place in $\mathit{normalize}_{\mathcal{Z}_+}$:

```
$\mathit{normalize}_{\mathcal{Z}_+}\ x\ P =$
$\mathbf{let}$
  $\mathit{nnf}\ \mathbf{as}\ \text{'}P \leftrightarrow Q\text{'} = \mathit{thm}\ (\mathit{decomp\_nnf}\ (\mathit{proveL}\ x))\ P$
  $\mathit{ac}\ \mathbf{as}\ \text{'}Q \leftrightarrow R\text{'} = \mathit{thm}\ (\mathit{decomp\_ac}\ x\ (\mathit{termlcm}\ x\ Q))\ Q$
$\mathbf{in}\ \mathit{fwd}\ \mathit{trans}\ [\mathit{fwd}\ \mathit{cong}_{\exists}\ [\mathit{gen}\ x\ (\mathit{fwd}\ \mathit{trans}\ [\mathit{nnf}, \mathit{ac}])], \mathit{unitcoeff}\ [R, l]]$
```

The auxiliary function *termlcm* computes l the *lcm* of all coefficients of x in Q and returns a function $\mathit{ldiv}_{>}$, which given c returns a theorem $k > 0$, where k is the result of dividing l by c. Its implementation is trivial and thus not shown.

In the last line of $\mathit{normalize}_{\mathcal{Z}_+}$ we cheat a bit to avoid excessive technicalities. For a start, we assume that in R all occurrences of the quantified variable are $l{\cdot}x$, whereas in reality they are $k{\cdot}c{\cdot}x$, where $k{\cdot}c = l$ is easily proved. As a result, x is indeed multiplied by l everywhere. The second cheat is that we have taken the liberty to employ a simple form of higher-order matching: matching the pattern $R(l{\cdot}x)$, where x is considered bound, against a formula f succeeds if and only if all occurrences of x in f are of the form $l{\cdot}x$, in which case function R is the result of λ-abstracting over all the occurrences of $l{\cdot}x$. Although on the programming language level formulae are a first-order data type and do not directly support higher-order matching, this simple instance of it is readily implemented.

3.3 Proof synthesis for Cooper's algorithm

The first step to prove (2) is to generalize it from the specific δ, $P_{-\infty}$ and $\mathcal{B}_P$ to arbitrary ones subject to certain assumptions:

$$\begin{aligned}
\mathit{cooper}_{-\infty} : \llbracket\ & 0 < \delta; \\
& \exists z. \forall x < z. P(x) \leftrightarrow Q(x); \\
& \forall x, k. Q(x) \leftrightarrow Q(x - k{\cdot}\delta); \\
& \forall x. \neg(\exists j \in [\delta], b \in B. x = b + j) \rightarrow P(x) \rightarrow P(x - \delta)\ \rrbracket \\
\Longrightarrow\ & \exists x. P(x) \leftrightarrow (\exists j \in [\delta]. Q(j) \vee \exists j \in [\delta], b \in B. P(b + j))
\end{aligned}$$

For the synthesis of the premises we use two functions:

delta computes δ as in (1) and returns δ, a theorem $0 < \delta$ and a function dvd_δ that, given a divisor d of δ, returns the theorem $d \mid \delta$.
bset computes $\mathcal{B}_P$ and returns B, its representation, and a function inB that, given a term $t \in \mathcal{B}_P$ returns a theorem $t \in B$.

The implementations of *delta* and *bset* are simple and omitted. But there is one optimization that is worth pointing out. For this we focus on dvd_δ. Let $\delta = \mathit{lcm}\{d_1, \ldots, d_n\}$. For each occurrence of one of the d_i we need the theorem $d_i \mid \delta$. To avoid recomputation, *delta* creates a mapping from the d_i to the theorems $d_i \mid \delta$ and dvd_δ simply looks up the relevant theorem. We have applied the same optimization to various functions throughout the paper.

Now we show how to prove the last three premises of $\mathit{cooper}_{-\infty}$, which correspond to (4),(5) and (6) respectively.

Derivation of $\exists z. \forall x < z. P(x) \leftrightarrow P_{-\infty}(x)$ First we prove the following theorems:

$(4)_\Diamond$: $\llbracket \exists z_1. \forall x < z_1. P_1(x) \leftrightarrow P_2(x); \exists z_2. \forall x < z_2. Q_1(x) \leftrightarrow Q_2(x) \rrbracket$
$\Longrightarrow \exists z. \forall x < z. (P_1(x) \Diamond Q_1(x)) \leftrightarrow (P_2(x) \Diamond Q_2(x))$, for $\Diamond \in \{\wedge, \vee\}$
$(4)_=$: $\exists z. \forall x < z. (x = t) \leftrightarrow \mathit{False}$ $\quad (4)_>$: $\exists z. \forall x < z. t < x \leftrightarrow \mathit{False}$

$(4)_{<}$: $\exists z.\forall x < z.x < t \leftrightarrow \mathit{True}$ $\quad$ $(4)_{(N)}$: $\exists z.\forall x < z.P \leftrightarrow P$
$(4)_{\bowtie}$: $\exists z.\forall x < z.(d \bowtie x + t) \leftrightarrow (d \bowtie x + t)$, for $\bowtie \in \{|, \nmid\}$

Instances of (4) are derived by $\mathit{prove}_{(4)}$:

```
$\mathit{decomp}_{(4)}\ x\ P =$
$\textbf{case}\ P\ \textbf{of}$
  $F \Diamond F \mid \Diamond \in \{\wedge\vee\} \Rightarrow ([F, G], \mathit{fwd}\ (4)_{\Diamond})$
  $t < y \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ (4)_{>}[t])$
  $y < t \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ (4)_{<}[t])$
  $y = t \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ (4)_{=}[t])$
  $d \bowtie y + t \mid y = x, \bowtie \in \{|, \nmid\} \Rightarrow ([], \boldsymbol{\lambda}[].\ (4)_{\bowtie}[t])$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ (4)_{(N)}[P])$

$\mathit{prove}_{(4)}\ x\ P = \mathit{thm}\ (\mathit{decomp}_{(4)}\ x)\ P$
```

Derivation of $\boldsymbol{\forall x, k.P_{-\infty}(x) \leftrightarrow P_{-\infty}(x - k{\cdot}\delta)}$ First we prove the following theorems:

$(5)_{\Diamond}$: $\llbracket \forall x, k.P(x) \leftrightarrow P(x - k{\cdot}\delta);\ \forall x, k.Q(x) \leftrightarrow Q(x - k{\cdot}\delta) \rrbracket$
$\Longrightarrow \forall x, k.(P(x) \Diamond Q(x)) \leftrightarrow (P(x - k{\cdot}\delta) \Diamond Q(x - k{\cdot}\delta))$ for $\Diamond \in \{\wedge, \vee\}$.
$(5)_{\bowtie}$: $d \mid \delta \Longrightarrow \forall x, k.(d \bowtie x + t) \leftrightarrow (d \bowtie x - k{\cdot}\delta + t)$, for $\bowtie \in \{|, \nmid\}$
$(5)_{(N)}$: $\forall x, k.\ P \leftrightarrow P$

Instances of (5) are derived by $\mathit{prove}_{(5)}$:

```
$\mathit{decomp}_{(5)}\ x\ \mathit{dvd}_{\delta}\ P =$
$\textbf{case}\ P\ \textbf{of}$
  $F \Diamond G \mid \Diamond \in \{\wedge, \vee\} \Rightarrow ([F, G], \mathit{fwd}\ (5)_{\Diamond})$
  $t < x \Rightarrow ([], \boldsymbol{\lambda}[].\ (5)_{(N)}[\mathit{False}])$
  $c < t \Rightarrow ([], \boldsymbol{\lambda}[].\ (5)_{(N)}[\mathit{True}])$
  $x = t \Rightarrow ([], \boldsymbol{\lambda}[].\ (5)_{(N)}[\mathit{False}])$
  $d \bowtie x + t \mid \bowtie \in \{|, \nmid\} \Rightarrow ([], \boldsymbol{\lambda}[].\ (\mathit{fwd}\ (5)_{\bowtie}\ [\mathit{dvd}_{\delta}\ d])[t])$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ (5)_{(N)}[P]);$

$\mathit{prove}_{(5)}\ x\ \mathit{dvd}_{\delta}\ P = \mathit{thm}\ (\mathit{decomp}_{(5)}\ x\ \mathit{dvd}_{\delta})\ P$
```

Derivation of $\boldsymbol{\forall x.\neg(\exists j \in [\delta], b \in \mathfrak{B}_P.x = b + j) \rightarrow P(x) \rightarrow P(x - \delta)}$ First we prove the following theorems:

$(6)_{\Diamond}$: $\llbracket \forall x.\neg(\exists j \in [\delta], b \in B.x = b + j) \rightarrow P(x) \rightarrow P(x - \delta);$
$\forall x.\neg(\exists j \in [\delta], b \in B.x = b + j) \rightarrow Q(x) \rightarrow Q(x - \delta) \rrbracket$
$\Longrightarrow \forall x.\neg(\exists j \in [\delta], b \in B.x = b + j)$
$\rightarrow (P(x) \Diamond Q(x)) \rightarrow (P(x - \delta) \Diamond Q(x - \delta))$, for $\Diamond \in \{\wedge, \vee\}$

$(6)_{(N)}$: $\forall x.\neg(\exists j \in [\delta], b \in B. x = b + j) \to F \to F$

$(6)_{<}$: $[\![0 < \delta]\!] \Longrightarrow \forall x.\neg(\exists j \in [\delta], b \in B. x = b + j) \to x < t \to x - \delta < t$

$(6)_{>}$: $[\![t \in B]\!] \Longrightarrow \forall x.\neg(\exists j \in [\delta], b \in B. x = b + j) \to t < x \to t < x - \delta$

$(6)_{=}$: $[\![0 < \delta; t - 1 \in B]\!]$
$\Longrightarrow \forall x.\neg(\exists j \in [\delta], b \in B. x = b + j) \to x = t \to (x - \delta) = t$

$(6)_{\bowtie}$: $[\![d \mid \delta]\!] \Longrightarrow$
$\forall x.\neg(\exists j \in [\delta], b \in B. x = b + j) \to d \bowtie x + t \to d \bowtie (x - \delta) + t$, for $\bowtie \in \{|, \nmid\}$

Instances of (6) are derived by $\mathit{prove}_{(6)}$:

```
$\mathit{decomp}_{(6)}\ x\ (\delta, \delta_{>}, \mathit{dvd}_{\delta})\ (B, \mathit{inB})\ P =$
**case** $P$ **of**
  $F \lozenge G \mid \lozenge \in \{\wedge, \vee\} \Rightarrow ([F, G], \mathit{fwd}\ (6)_{\lozenge})$
  $t < y \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{fwd}\ ((6)_{>}[t, B, \delta])\ [\mathit{inB}\ t])$
  $y < t \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{fwd}\ ((6)_{<}[\delta, B, t])\ [\delta_{>}])$
  $y = t \mid y = x \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{fwd}\ ((6)_{=}[\delta, t, B])\ [\delta_{>}, \mathit{inB}\ t - 1])$
  $d \bowtie y + t \mid y = x, \bowtie \in \{|, \nmid\} \Rightarrow ([], \boldsymbol{\lambda}[].\ \mathit{fwd}\ ((6)_{\bowtie}[d, \delta, B, t])\ [\mathit{dvd}_{\delta}\ d])$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].\ (6)_{(N)}[\delta, B, P])$

$\mathit{prove}_{(6)}\ x\ (\delta, \delta_{>}, \mathit{dvd}_{\delta})\ (B, \mathit{inB})\ P\ =$
  $\mathit{thm}\ (\mathit{decomp}_{(6)}\ x\ (\delta, \delta_{>}, \mathit{dvd}_{\delta})\ (B, \mathit{inB}))\ P$
```

The overall proof Quantifier elimination for $\mathcal{Z}_{+}$ is finally implemented by $\mathit{qelim}_{\mathcal{Z}_{+}}$.

```
$\mathit{prove}_{\delta\infty\mathcal{B}}\ x\ P\ =$
**let**
  $(\delta, \delta_{>}, \mathit{dvd}_{\delta}) = \mathit{delta}\ x\ P$
  $(B, \mathit{inB}) = \mathit{bset}\ x\ P$
  $\mathit{th}_{(4)} = \mathit{prove}_{(4)}\ x\ P$
  $\mathit{th}_{(5)} = \mathit{prove}_{(5)}\ x\ \mathit{dvd}_{\delta}\ P$
  $\mathit{th}_{(6)} = \mathit{prove}_{(6)}\ x\ (\delta, \delta_{>}, \mathit{dvd}_{\delta})\ (B, \mathit{inB})\ P$
**in** $[\delta_{>}, \mathit{th}_{(4)}, \mathit{th}_{(5)}, \mathit{th}_{(6)}]$

$\mathit{cooper}\ \exists x.P\ =$
**let**
  $\mathit{nth}$ **as** '$\exists x.P \leftrightarrow \exists x.Q$' $= \mathit{normalize}_{\mathcal{Z}_{+}}\ x\ P$
  $\mathit{cpth} = \mathit{fwd}\ \mathit{cooper}_{-\infty}\ (\mathit{prove}_{\delta\infty\mathcal{B}}\ x\ Q)$
**in** $\mathit{fwd}\ \mathit{trans}\ [\mathit{nth}, \mathit{expand}_{\exists}\ \mathit{cpth}]$

$\mathit{qelim}_{\mathcal{Z}_{+}}\ = \mathit{qelim}\ \mathit{cooper}$
```

The function $\mathit{prove}_{\delta\infty\mathcal{B}}$ returns exactly the list of premises of $\mathit{cooper}_{-\infty}$. For efficiency it should traverse the formula only once and synthesize (4),

(5) and (6) at once. This is easy to achieve by "merging" the different $\mathit{decomp}_{(.)}$ functions. At the end we apply a function $\mathit{expand}_{\exists}$ which takes some theorem and returns an equivalent one where all occurrences of $\exists x \in I$, where I is some finite set, have been expanded into finite disjunctions. Typically this is performed by rewriting and we do not discuss the details. We assume that at the same time rewriting also evaluates all ground arithmetic and logical expressions.

3.4 Proof synthesis for Ferrante and Rackoff's algorithm

To derive an instance of Theorem 2 we first generalize it from the specific $\mathcal{U}_P, P_{-\infty}$ and $P_{+\infty}$ to arbitrary ones subject to certain assumptions:

$$\begin{aligned}
\mathit{fr} :& \llbracket \mathsf{finite}\ U\ ;\ \exists y.\forall x < y.P(x) \leftrightarrow Q\ ;\ \exists y.\forall x > y.P(x) \leftrightarrow R\ ;\\
&\forall x.\neg Q \wedge P(x) \rightarrow \exists l \in U.l \leq x\ ;\ \forall x.\neg R \wedge P(x) \rightarrow \exists u \in U.x \leq u\ ;\\
&\forall x, l, u.(\forall y.l<y<u \rightarrow y \notin U) \wedge l<x<u \wedge P(x) \rightarrow \forall y.l<y<u \rightarrow P(y)\rrbracket\\
&\Longrightarrow \exists x.P(x) \leftrightarrow (Q \vee R \vee \exists(u,u') \in U^2.P(\frac{u+u'}{2}))
\end{aligned}$$

In order to prove the premises of *fr*, we use a function *uset*, analogous to *bset*, which computes the set $\mathcal{U}_P$ and returns a theorem $\mathsf{finite}\ U$ and a function *inU* which, given a $t \in \mathcal{U}_P$, returns a theorem $t \in U$, where U represents $\mathcal{U}_P$.

The derivation of the premises follows Cooper's algorithm. In fact the second premise is derived by $\mathit{prove}_{(4)}$, the third by its (omitted) dual $\mathit{prove}_{(7)}$. The derivation of the fourth and the fifth premise are dual and we only show the synthesis of $\forall x.\neg P_{-\infty} \wedge P(x) \rightarrow \exists l \in \mathcal{U}_P.l \leq x$, which is the result of $\mathit{prove}_{(8)}$. First we prove a few easy lemmas:

$$\begin{aligned}
&(8)_{\Diamond}\!: \llbracket \forall x.\neg P' \wedge P(x) \rightarrow \exists l \in U.l \leq x; \forall x.\neg Q' \wedge Q(x) \rightarrow \exists l \in U.l \leq x\rrbracket\\
&\quad\Longrightarrow \forall x.\neg(P' \Diamond Q') \wedge (P(x) \Diamond Q(x)) \rightarrow \exists l \in U.l \leq x,\ \text{for}\ \Diamond \in \{\wedge, \vee\}\\
&(8)_{<}:\ \forall x.\neg \mathit{True} \wedge x < t \rightarrow \exists l \in U.l \leq x\\
&(8)_{\bowtie}:\ \llbracket t \in U\rrbracket \Longrightarrow \forall x.\neg \mathit{False} \wedge t \bowtie x \rightarrow \exists l \in U.l \leq x, \text{for} \bowtie \in \{=, >\}\\
&(8)_{(N)}:\ \forall x.\neg F \wedge F \rightarrow \exists l \in U.l \leq x.
\end{aligned}$$

Function $\mathit{prove}_{(8)}$ performs the derivation:

```
$\mathit{decomp}_{(8)}\ x\ \mathit{inU}\ P =$
**case** $P$ **of**
  $F \Diamond G \mid \Diamond \in \{\wedge, \vee\} \Rightarrow ([F, G], \mathit{fwd}\ (8)_{\Diamond})$
  $y < t \mid y = x\} \Rightarrow ([], \boldsymbol{\lambda}[].(8)_{<}[t])$
  $y \bowtie t \mid y = x, \bowtie \in \{=, >\} \Rightarrow ([], \boldsymbol{\lambda}[].\mathit{fwd}\ (8)_{\bowtie}\ [\mathit{inU}\ t])$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].(8)_{(N)}[P])$

$\mathit{prove}_{(8)}\ x\ \mathit{inU}\ P = \mathit{thm}\ (\mathit{decomp}_{(8)}\ x\ \mathit{inU})\ P$
```

The derivation of the last premise of *fr* is the result of $\mathit{prove}_{(10)}$, which uses the following theorems:

$$\begin{aligned}
(10)_{\Diamond} : & [\![\forall x, l, u.(\forall y.l < y < u \to y \notin U) \wedge l < x < u \wedge P(x) \\
& \to \forall y.l < y < u \to P(y); \\
& \forall x, l, u.(\forall y.l < y < u \to y \notin U) \wedge l < x < u \wedge Q(x) \\
& \to \forall y.l < y < u \to Q(y)]\!] \Longrightarrow \\
& \forall x, l, u.(\forall y.l < y < u \to y \notin U) \wedge l < x < u \wedge P(x) \Diamond Q(x) \\
& \to \forall y.l < y < u \to P(y) \Diamond Q(y), \text{ for } \Diamond \in \{\wedge, \vee\} \\
(10)_{\bowtie} : & t \in U \Longrightarrow \forall x, l, u.(\forall y.l < y < u \to y \notin U) \wedge l < x < u \wedge x \bowtie t \\
& \to \forall y.l < y < u \to y \bowtie t, \text{for } \bowtie \in \{=, <, >\} \\
(10)_{(N)} : & \forall x, l, u.(\forall y.l < y < u \to y \notin U) \wedge l < x < u \wedge P \to \forall y.l < y < u \to P
\end{aligned}$$

```
$\mathit{decomp}_{(10)}\ x\ \mathit{inU}\ P =$
**case** $P$ **of**
  $F \Diamond G \mid \Diamond \in \{\wedge, \vee\} \Rightarrow ([F, G], \mathit{fwd}\ (10)_{\Diamond})$
  $y \bowtie t \mid y = x, \bowtie \in \{=, <, >\} \Rightarrow ([], \boldsymbol{\lambda}[].\mathit{fwd}\ (10)_{\bowtie}\ [\mathit{inU}\ t])$
  $\_ \Rightarrow ([], \boldsymbol{\lambda}[].(10)_{(N)}[P])$

$\mathit{prove}_{(10)}\ x\ \mathit{inU}\ P = \mathit{thm}\ (\mathit{decomp}_{(10)}\ x\ \mathit{inU})\ P$
```

We merge all the functions that derive the premises into $\mathit{prove}_{\pm\infty\mathcal{U}}$, which returns a 6-element list corresponding to the premises of *fr*. The overall quantifier elimination procedure for $\mathcal{R}_+$ is implemented by $\mathit{qelim}_{\mathcal{R}_+}$.

```
$\mathit{ferrack}\ \exists x.P\ =$
**let** $\mathit{nth}$ **as** '$\exists x.P \leftrightarrow \exists x.Q$' $= \mathit{normalize}_{\mathcal{R}_+}\ x\ P$
**in** $\mathit{expand}_{\exists}(\mathit{fwd\ trans}\ [\mathit{nth}, \mathit{fwd\ fr}\ (\mathit{prove}_{\pm\infty\mathcal{U}}\ x\ Q)])$

$\mathit{qelim}_{\mathcal{R}_+}\ =\ \mathit{qelim\ ferrack}$
```

4 Reflection

After a simplified explanation of reflection we show its application to both Cooper's algorithm and the one by Ferrante and Rackoff. The content of this section is a minimally revised version of [9] and [10].

4.1 An informal introduction to reflection

Reflection means to perform a proof step by computation inside the logic rather than inside some external programming language. The latter is traditionally referred to as a *meta-language* (ML) and is what we have relied on so far. Inside the logic it is not possible to write functions by pattern matching over the syntax because two syntactically distinct formulae may be logically equivalent. Hence the relevant fragment of formulae must be represented (*reflected*) inside the logic as a datatype. Sometimes this datatype is called the *shadow syntax* [30]. We call it *rep*, the representation.

The two levels of formulae must be connected by two functions:

interp, a function in the logic, maps an element of *rep* to the formula it represents, and

reify, a function in ML, maps a formula to its representation.

The two functions should be inverses of each other. Informally $\mathsf{interp}(\mathit{reify}\ P) \leftrightarrow P$ should hold. More precisely, taking the ML representation of a formula P and applying *reify* to it yields an ML representation of a term p of type *rep* such that $\mathsf{interp}\ p \leftrightarrow P$ holds.

Typically, the formalized proof step is some equivalence $P \leftrightarrow P'$ where P is given and P' is some simplified version of P (e.g. the elimination of quantifiers). This transformation is now expressed as a recursive function *simp* of type $\mathit{rep} \to \mathit{rep}$. We prove (typically by induction on *rep*) that *simp* preserves the interpretation:

$$\mathsf{interp}\ p \leftrightarrow \mathsf{interp}(\mathit{simp}\ p)$$

To apply this theorem to a given formula P we proceed as follows:

1. Create a *rep*-term p from P using *reify*. This is the *reification* step which must be performed in ML.
2. Prove $P \leftrightarrow \mathsf{interp}\ p$. Usually this is trivial by rewriting with the definition of interp.
3. Instantiate *simp*'s correctness theorem above, compute $p' = \mathit{simp}\ p$ and obtain the theorem $\mathsf{interp}\ p \leftrightarrow \mathsf{interp}\ p'$. This is the *evaluation* step.
4. Simplify $\mathsf{interp}\ p'$, again by rewriting with the definition of interp, yielding a theorem $\mathsf{interp}\ p' \leftrightarrow P'$

The final theorem $P \leftrightarrow P'$ holds by transitivity.

The evaluation step is crucial for efficiency as all other steps are typically linear-time. In our work we employ Isabelle's built-in ground term evaluator [2] which generates executable code from the definition of *simp* and “runs” *simp p*. The fact that the executable code is again ML is a coincidence. A similar approach is adopted in PVS [15] and in ACL2 [7]. Other approaches include the the use of an internal λ-calculus evaluator [27] as in Coq [5]. One could also perform the evaluation step by rewriting inside the theorem prover, but the performance penalty is usually prohibitive.

There is also the practical issue of where *reify* comes from. In general, the implementor of the reflected proof procedure must program it in ML and link it into the above chain of deductions. But because *reify* must be the inverse of interp, it is often possible to automate this step. Isabelle implements a sufficiently general inversion scheme for interp (which even covers bound variables represented by de Bruijn indices) such that for all of the examples in this paper, *reify* is dealt with automatically.

4.2 Reflecting linear arithmetic

In this section we represent linear arithmetic formulae inside the logic. We also give a generic quantifier elimination, sketch normalization of formulae and present the common parts of both algorithms.

Terms and Formulae We define the syntax of terms and formulae as follows:

$$\begin{aligned}
&\mathsf{datatype}\ \tau = \widehat{\alpha} \mid \boldsymbol{v}_{nat} \mid \boldsymbol{-}\,\tau \mid \tau \boldsymbol{+} \tau \mid \tau \boldsymbol{-} \tau \mid \alpha \boldsymbol{*} \tau\\
&\mathsf{datatype}\ \phi = \boldsymbol{T} \mid \boldsymbol{F} \mid \tau \boldsymbol{<} \tau \mid \tau \boldsymbol{\le} \tau \mid \tau \boldsymbol{=} \tau \mid \tau \boldsymbol{\neq} \tau \mid int \boldsymbol{\mid} \tau \mid int \boldsymbol{\nmid} \tau\\
&\qquad\qquad \mid \boldsymbol{\neg}\,\phi \mid \phi \boldsymbol{\wedge} \phi \mid \phi \boldsymbol{\vee} \phi \mid \phi \boldsymbol{\rightarrow} \phi \mid \phi \boldsymbol{\leftrightarrow} \phi \mid \boldsymbol{\exists}\,\phi \mid \boldsymbol{\forall}\,\phi
\end{aligned}$$

In the logic, datatypes are declared using datatype. Lists are built up from the empty list [] and consing $\cdot$; the infix @ appends two lists. For a list l, $\{\!\{l\}\!\}$ denotes the set of elements of l, and $l!n$ denotes its n^{th} element. For a finite set S, $|S|$ denotes the cardinality of S.

The constant c in the logic is represented by the term $\widehat{c}$. Constants are integers for $\mathcal{Z}_+$, i.e. $\alpha = int$, and rational numbers for $\mathcal{R}_+$, i.e. $\alpha = rat$. That is, τ and ϕ are really parameterized by type α. Bound variables are represented by de Bruijn indices: $\boldsymbol{v}_n$ represents the bound variable with index n (a natural number). Hence quantifiers need not carry variable names. The bold symbols $\boldsymbol{+}$, $\boldsymbol{\le}$, $\boldsymbol{\wedge}$ etc are constructors and reflect their counterparts $+$, $\le$, $\wedge$ etc in the logic. We use $\boldsymbol{\bowtie}$ (resp. $\bowtie$) as a place-holder for $\boldsymbol{=}, \boldsymbol{\neq}, \boldsymbol{\le}$ or $\boldsymbol{<}$ (resp. $=, \neq, \le$ or $<$). We also use $\boldsymbol{\Diamond}$ (resp. $\Diamond$) as place-holder for $\boldsymbol{\wedge}, \boldsymbol{\vee}, \boldsymbol{\rightarrow}$ or $\boldsymbol{\leftrightarrow}$ (resp. $\wedge, \vee, \rightarrow, \leftrightarrow$). *In the following p and q (resp. s and t) are of type ϕ (resp. τ).*

$$
\begin{array}{ll}
\llparenthesis\widehat{c}\rrparenthesis_{\tau}^{vs} & = c \\
\llparenthesis\boldsymbol{v}_n\rrparenthesis_{\tau}^{vs} & = vs!n \\
\llparenthesis -\ t\rrparenthesis_{\tau}^{vs} & = -\llparenthesis t\rrparenthesis_{\tau}^{vs} \\
\llparenthesis t\ \boldsymbol{+}\ s\rrparenthesis_{\tau}^{vs} & = \llparenthesis t\rrparenthesis_{\tau}^{vs} + \llparenthesis s\rrparenthesis_{\tau}^{vs} \\
\llparenthesis t\ \boldsymbol{-}\ s\rrparenthesis_{\tau}^{vs} & = \llparenthesis t\rrparenthesis_{\tau}^{vs} - \llparenthesis s\rrparenthesis_{\tau}^{vs} \\
\llparenthesis c\ \boldsymbol{*}\ t\rrparenthesis_{\tau}^{vs} & = c{\cdot}\llparenthesis t\rrparenthesis_{\tau}^{vs}
\end{array}
\qquad
\begin{array}{ll}
\llparenthesis\boldsymbol{T}\rrparenthesis^{vs} & = \mathit{True} \\
\llparenthesis\boldsymbol{F}\rrparenthesis^{vs} & = \mathit{False} \\
\llparenthesis t \boldsymbol{\bowtie} s\rrparenthesis^{vs} & = (\llparenthesis t\rrparenthesis_{\tau}^{vs} \bowtie \llparenthesis s\rrparenthesis_{\tau}^{vs}) \\
\llparenthesis \boldsymbol{\neg} p\rrparenthesis^{vs} & = (\neg\llparenthesis p\rrparenthesis^{vs}) \\
\llparenthesis p \boldsymbol{\Diamond} q\rrparenthesis^{vs} & = (\llparenthesis p\rrparenthesis^{vs} \Diamond \llparenthesis q\rrparenthesis^{vs}) \\
\llparenthesis \boldsymbol{\exists}\, p\rrparenthesis^{vs} & = (\exists x. \llparenthesis p\rrparenthesis^{x\cdot vs}) \\
\llparenthesis \boldsymbol{\forall}\, p\rrparenthesis^{vs} & = (\forall x. \llparenthesis p\rrparenthesis^{x\cdot vs})
\end{array}
$$

Fig. 2. Semantics of the shadow syntax

The interpretation functions ($\llparenthesis.\rrparenthesis_{\tau}^{\cdot}$ and $\llparenthesis.\rrparenthesis^{\cdot}$) in Fig. 2 map the representations back into logic. They are parameterized by an environment vs which is a list of integer expressions for $\mathcal{Z}_+$, and a list of real expressions for $\mathcal{R}_+$. The de Bruijn index $\boldsymbol{v}_n$ picks out the n^{th} element from that list. Since $\mathcal{Z}_+$ includes divisibility relations, we define $\llparenthesis i \mid t\rrparenthesis^{vs} = (i \mid \llparenthesis t\rrparenthesis_{\tau}^{vs})$ and $\llparenthesis i \boldsymbol{\nmid} t\rrparenthesis^{vs} = i \nmid \llparenthesis t\rrparenthesis_{\tau}^{vs}$.

Example 4.

$$
\begin{gathered}
\llparenthesis \boldsymbol{\forall}\,\boldsymbol{\exists}\,\boldsymbol{\exists}(3 \boldsymbol{*} \boldsymbol{v}_0 \boldsymbol{+} 5 \boldsymbol{*} \boldsymbol{v}_1 \boldsymbol{-} \boldsymbol{v}_2 \boldsymbol{=} \widehat{0})\rrparenthesis^{[]} \leftrightarrow (\forall x. \exists y, z. 3{\cdot}z + 5{\cdot}y - x = 0) \\
\llparenthesis 3 \boldsymbol{*} \boldsymbol{v}_0 \boldsymbol{+} 5 \boldsymbol{*} \boldsymbol{v}_1 \boldsymbol{\le} \boldsymbol{v}_2\rrparenthesis^{[x,y,z]} \leftrightarrow (3{\cdot}x + 5{\cdot}y \le z)
\end{gathered}
$$

$$
\begin{array}{lcl}
\mathsf{qelim}\ qe\ (\boldsymbol{\forall}\, p) & = & \boldsymbol{\neg} qe \boldsymbol{\neg}(\mathsf{qelim}\ qe\ p) \\
\mathsf{qelim}\ qe\ (\boldsymbol{\exists}\, p) & = & qe(\mathsf{qelim}\ qe\ p) \\
\mathsf{qelim}\ qe\ (p \boldsymbol{\Diamond} q) & = & (\mathsf{qelim}\ qe\ p) \boldsymbol{\Diamond} (\mathsf{qelim}\ qe\ p) \\
\mathsf{qelim}\ qe\ p & = & p
\end{array}
$$

Fig. 3. Quantifier elimination for ϕ-formulae

Generic quantifier elimination Assume we have a function qe, that eliminates one $\boldsymbol{\exists}$ in front of quantifier-free ϕ-formulae. The function qelim in Fig. 3 applies qe to all quantified sub-formulae in a bottom-up fashion. Let $\mathsf{qfree}\ p$ formalize that p is quantifier-free . We automatically prove the main property (12) of qelim by structural induction on p: if qe takes a quantifier-free formula q and returns a quantifier-free formula q' equivalent to $\boldsymbol{\exists}\, q$, then $\mathsf{qelim}\ qe$ is a quantifier-elimination procedure:

$$
\begin{gathered}
(\forall vs, q.\ \mathsf{qfree}\ q \longrightarrow \mathsf{qfree}\ (qe\ q) \wedge (\llparenthesis qe\ q\rrparenthesis^{vs} \leftrightarrow \llparenthesis \boldsymbol{\exists}\, q\rrparenthesis^{vs})) \\
\longrightarrow \mathsf{qfree}\ (\mathsf{qelim}\ qe\ p) \wedge (\llparenthesis \mathsf{qelim}\ qe\ p\rrparenthesis^{vs} \leftrightarrow \llparenthesis p\rrparenthesis^{vs}).
\end{gathered}
\tag{12}
$$

The implementation of qelim in Fig. 3 is naive and only used for the sake of presentation. In reality, the definition involves several simplifications such as distribution of $\exists$ over $\vee$, lazy evaluation of $\wedge, \vee, \longrightarrow$ and $\longleftrightarrow$, etc.

In §4.3 and §4.4 we present cooper and $\mathsf{ferrack}$, two instances of qe satisfying the premise of (12) for $\mathcal{Z}_+$ and $\mathcal{R}_+$, respectively.

Normalized formulae and elimination sets Normalized ϕ-formulae are defined via predicates $\mathsf{isnorm}_{\mathcal{Z}_+}$ and $\mathsf{isnorm}_{\mathcal{R}_+}$ for $\mathcal{Z}_+$ and $\mathcal{R}_+$ respectively. The definition of $\mathsf{isnorm}_{\mathcal{Z}_+}$ is as follows:

$$\begin{array}{lcll}
\mathsf{isnorm}_{\mathcal{Z}_+}\ (p \Diamond q) & = & (\mathsf{isnorm}_{\mathcal{Z}_+}\ l\ p) \wedge (\mathsf{isnorm}_{\mathcal{Z}_+}\ l\ q) & \text{for } \Diamond \in \{\wedge, \vee\} \\
\mathsf{isnorm}_{\mathcal{Z}_+}\ (\boldsymbol{v}_0 \bowtie t) & = & \mathsf{unbound}_{\tau}\ t & \\
\mathsf{isnorm}_{\mathcal{Z}_+}\ (t \bowtie \boldsymbol{v}_0) & = & \mathsf{unbound}_{\tau}\ t & \text{for } \bowtie \in \{<, \leq\} \\
\mathsf{isnorm}_{\mathcal{Z}_+}\ (d \bowtie \boldsymbol{v}_0 + t) & = & d > 0 \wedge \mathsf{unbound}_{\tau}\ t & \text{for } \bowtie \in \{\mid, \nmid\} \\
\mathsf{isnorm}_{\mathcal{Z}_+}\ p & = & \mathsf{unbound}_{\phi}\ p &
\end{array}$$

The predicates $\mathsf{unbound}_{\tau}$ and $\mathsf{unbound}_{\phi}$ formalize that a τ-term and a ϕ-formula, respectively, do not involve $\boldsymbol{v}_0$. The definition of $\mathsf{isnorm}_{\mathcal{R}_+}$ just excludes the $\mid$ and $\nmid$ atoms. Normalization is performed as explained in §2.2 by the functions $\mathsf{norm}_{\mathcal{Z}_+}$ and $\mathsf{norm}_{\mathcal{R}_+}$ satisfying (13) and (14).

$$\mathsf{qfree}\ p \longrightarrow \mathsf{isnorm}_{\mathcal{Z}_+}(\mathsf{norm}_{\mathcal{Z}_+}\ p) \wedge (\!|\exists(\mathsf{norm}_{\mathcal{Z}_+}\ p)|\!)^{vs} \leftrightarrow (\!|\exists\, p|\!)^{vs} \qquad (13)$$

$$\mathsf{qfree}\ p \longrightarrow \mathsf{isnorm}_{\mathcal{R}_+}(\mathsf{norm}_{\mathcal{R}_+}\ p) \wedge (\!|\exists(\mathsf{norm}_{\mathcal{R}_+}\ p)|\!)^{vs} \leftrightarrow (\!|\exists\, p|\!)^{vs} \qquad (14)$$

Fig. 4 shows the reflected counterparts of the functions defined in Fig. 1. If P is reflected by p then $P_{-\infty}$ and $P_{+\infty}$ are reflected by p_- and p_+, and $\mathcal{U}_P, \mathcal{B}_P$ and $\mathcal{A}_P$ are reflected by the lists $\mathsf{U}\ p, \mathsf{B}\ p$ and $\mathsf{A}\ p$.

p	$\mathsf{U}\ p$	$\mathsf{B}\ p$	$\mathsf{A}\ p$	p_-	p_+
$q \Diamond r$	$\mathsf{U}\ q$ @ $\mathsf{U}\ r$	$\mathsf{B}\ q$ @ $\mathsf{B}\ r$	$\mathsf{A}\ q$ @ $\mathsf{A}\ r$	$q_- \Diamond r_-$	$q_+ \Diamond r_+$
$t < \boldsymbol{v}_0$	$[t]$	$[t]$	$[]$	$\boldsymbol{F}$	$\boldsymbol{T}$
$\boldsymbol{v}_0 < t$	$[t]$	$[]$	$[t]$	$\boldsymbol{T}$	$\boldsymbol{F}$
$t \leq \boldsymbol{v}_0$	$[t]$	$[t - \widehat{1}]$	$[]$	$\boldsymbol{F}$	$\boldsymbol{T}$
$\boldsymbol{v}_0 \leq t$	$[t]$	$[]$	$[t + \widehat{1}]$	$\boldsymbol{T}$	$\boldsymbol{F}$
$\boldsymbol{v}_0 = t$	$[t]$	$[t - \widehat{1}]$	$[t + \widehat{1}]$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{v}_0 \neq t$	$[t]$	$[t]$	$[t]$	$\boldsymbol{T}$	$\boldsymbol{T}$
_	$\emptyset$	$\emptyset$	$\emptyset$	p	p

Fig. 4. Definition of $\mathsf{U}\ p$, $\mathsf{B}\ p$, $\mathsf{A}\ p$, p_- and p_+

Our reflective implementation represents all sets by lists. Because the complexity of both algorithms depends highly on the size of the elimination sets (see Theorems 1 and 2), our implementation removes duplicate elements from the elimination sets via function $\mathsf{remdups}$. This function compares terms for semantic equality by linearizing them first. We explain this process in some detail.

A τ-term t is *linear* ($\mathsf{islin}_\tau\ t$) if it has the form

$$c_1 * \boldsymbol{v}_{i_1} + \cdots + c_n * \boldsymbol{v}_{i_n} + \widehat{c_{n+1}}$$

where $n \in \mathbb{N}, i_1 < \cdots < i_n$ and $\forall j \leq n. c_j \neq 0$.

$$\begin{array}{lcl}
\mathsf{islin'}_\tau\ n_0\ \widehat{i} & = & \mathit{True} \\
\mathsf{islin'}_\tau\ n_0\ (i * \boldsymbol{v}_n + r) & = & i \neq 0 \wedge n_0 \leq n \wedge \mathsf{islin'}_\tau\ (n+1)\ r \\
\mathsf{islin'}_\tau\ n_0\ t & = & \mathit{False} \\
\mathsf{islin}_\tau\ t & = & \mathsf{islin'}_\tau\ 0\ t
\end{array}$$

An arbitrary τ-term t is linearized by function $\mathsf{lin}_\tau\ t$ (definition omitted), which replaces every $+, -, *$ and $-$ by $-^l, +_l, -_l, *_l$ and $-^l$. The interpretation of linear terms is preserved by this process (theorems for $-_l$ and $-^l$ are omitted):

$$\mathsf{islin}_\tau\ t \wedge \mathsf{islin}_\tau\ s \longrightarrow ((⦇t +_l s⦈^{vs}_\tau = ⦇t + s⦈^{vs}_\tau) \wedge \mathsf{islin}_\tau\ (t +_l s)) \quad (15)$$

$$\mathsf{islin}_\tau\ t \longrightarrow ((⦇c *_l t⦈^{vs}_\tau ⦇c * t⦈^{vs}_\tau) \wedge \mathsf{islin}_\tau (c *_l t) \quad (16)$$

The proofs and definitions of $+_l$ and $*_l$ are simple. The key clauses in the definition are shown below.

```
$(k * \boldsymbol{v}_n + r) +_l (l * \boldsymbol{v}_m + s) =$
$\mathbf{if}\ n = m\ \mathbf{then}$
  $\mathbf{if}\ k + l = 0\ \mathbf{then}\ r +_l s\ \mathbf{else}\ (k + l) * \boldsymbol{v}_n + (r +_l s)$
$\mathbf{else\ if}\ n \leq m\ \mathbf{then}\ k * \boldsymbol{v}_n + (r +_l (l * \boldsymbol{v}_m + s)$
  $\mathbf{else}\ l * \boldsymbol{v}_m + (k * \boldsymbol{v}_n + r) +_l s$
$(k * \boldsymbol{v}_n + r) +_l \widehat{b} = k * \boldsymbol{v}_n + (r +_l \widehat{b})$
$\widehat{a} +_l (l * \boldsymbol{v}_n + s) = l * \boldsymbol{v}_n + (s +_l \widehat{a})$
$\widehat{k} +_l \widehat{l} = \widehat{k + l}$

$c *_l \widehat{i} = \widehat{c \cdot i}$
$c *_l (i * \boldsymbol{v}_n + r) = \mathbf{if}\ c = 0\ \mathbf{then}\ \widehat{0}\ \mathbf{else}\ (c \cdot i) * \boldsymbol{v}_n + (c *_l r)$

$-^l\ t = -1 *_l t$
$t -_l s = t +_l (-^l\ s)$
```

Using (15) and (16) we automatically prove lin_τ's main property:

$$(⦇\mathsf{lin}_\tau\ t⦈^{vs}_\tau = ⦇t⦈^{vs}_\tau) \wedge \mathsf{islin}_\tau\ (\mathsf{lin}_\tau\ t). \quad (17)$$

4.3 Reflecting Cooper's algorithm

In this section we present a verified implementation of Cooper's algorithm which differs only slightly from [10].

Cooper's theorem Now we prove the analogue of $cooper_{-\infty}$ for ϕ-formulae:

$$\begin{aligned}\mathsf{isnorm}_{\mathcal{Z}_+}\ p \longrightarrow (\llparenthesis\exists\, p\rrparenthesis^{vs} \longleftrightarrow ((\exists j \in \{1..\mathsf{D}_p\}.\llparenthesis p_-\rrparenthesis^{j\cdot vs}) \\ \vee (\exists j \in \{1..\mathsf{D}_p\}, b \in \{\llparenthesis t\rrparenthesis_{\tau}^{i\cdot vs} \mid t \in \{\!\{\mathsf{B}\ p\}\!\}\}.\llparenthesis p\rrparenthesis^{(b+j)\cdot vs}))).\end{aligned} \tag{18}$$

Note that the i in $\llparenthesis t\rrparenthesis_{\tau}^{i\cdot vs}$ is free because t does not depend on $\boldsymbol{v}_0$.

In Fig. 5 we define a function D_p to compute the δ in (1) and a predicate alldvd_{ϕ} to express δ's main property (19), which we prove automatically. It is the predicate alldvd_{ϕ} we use, when during induction we need that $d \mid \delta$ for an atom $d \mid \boldsymbol{v}_0 + t$. Concretely, we prove properties involving D_p for a general l satisfying $\mathsf{alldvd}_{\phi}\ l\ p$.

$$\mathsf{isnorm}_{\mathcal{Z}_+}\ p \longrightarrow \mathsf{D}_p > 0 \wedge \mathsf{alldvd}_{\phi}\ \mathsf{D}_p\ p \tag{19}$$

p	$\mathsf{D}\ p$	$\mathsf{alldvd}_{\phi}\ l\ p$
$q \lozenge r$	$lcm\ (\mathsf{D}\ q)\ (\mathsf{D}\ r)$	$(\mathsf{alldvd}_{\phi}\ l\ q) \wedge (\mathsf{alldvd}_{\phi}\ l\ r)$
$d \mid \boldsymbol{v}_0 + t$	d	$d \mid l$
$d \nmid \boldsymbol{v}_0 + t$	d	$d \mid l$
_	1	*True*

Fig. 5. D_p and alldvd_{ϕ} for normalized ϕ-formulae

As in §3.3 we prove the (reflected) premises of $cooper_{-\infty}$:

$$\mathsf{isnorm}_{\mathcal{Z}_+}\ p \longrightarrow \exists z.\forall x.x < z \longrightarrow (\llparenthesis p\rrparenthesis^{x\cdot vs} = \llparenthesis p_-\rrparenthesis^{x\cdot vs}) \tag{20}$$

$$\mathsf{isnorm}_{\mathcal{Z}_+}\ p \longrightarrow \forall x, k.\llparenthesis p_-\rrparenthesis^{x\cdot vs} = \llparenthesis p_-\rrparenthesis^{(x-k\cdot\mathsf{D}_p)\cdot vs} \tag{21}$$

$$\begin{aligned}\mathsf{isnorm}_{\mathcal{Z}_+}\ p \longrightarrow \forall x.\neg(\exists j \in \{\!\{1..\mathsf{D}_p\}\!\}.\exists b \in \{\llparenthesis t\rrparenthesis_{\tau}^{i\cdot vs} \mid t \in \{\!\{\mathsf{B}\ p\}\!\}\}.\llparenthesis p\rrparenthesis^{(b+j)\cdot vs}) \\ \longrightarrow \llparenthesis p\rrparenthesis^{x\cdot vs} \longrightarrow \llparenthesis p\rrparenthesis^{(x-\delta_p)\cdot vs}\end{aligned} \tag{22}$$

All these theorems are proved by structural induction on p just as in §2.3, but now once and for all rather than for each instance. With these theorems and $cooper_{-\infty}$ we easily prove Cooper's theorem for ϕ-formulae.

$$
\begin{array}{lcll}
\mathsf{mirror}\ (p \Diamond q) & = & (\mathsf{mirror}\ l\ p) \Diamond (\mathsf{mirror}\ l\ q) & \text{for } \Diamond \in \{\wedge, \vee\} \\
\mathsf{mirror}\ (\boldsymbol{v}_0 \bowtie t) & = & \boldsymbol{v}_0 \bowtie (-\ t) & \text{for } \bowtie \in \{=, \neq\} \\
\mathsf{mirror}\ (\boldsymbol{v}_0 \bowtie t) & = & (-\ t) \bowtie \boldsymbol{v}_0 & \text{for } \bowtie \in \{<, \leq\} \\
\mathsf{mirror}\ (t \bowtie \boldsymbol{v}_0) & = & \boldsymbol{v}_0 \bowtie (-\ t) & \text{for } \bowtie \in \{<, \leq\} \\
\mathsf{mirror}\ (d \bowtie \boldsymbol{v}_0 + r) & = & d \bowtie \boldsymbol{v}_0 + (-\ r) & \text{for } \bowtie \in \{\mid, \nmid\} \\
\mathsf{mirror}\ p & = & p &
\end{array}
$$

Fig. 6. Duality principle for ϕ-formulae.

A duality principle We formalize the duality principle ($P_{-\infty}$ and $\mathcal{B}_P$ vs. $P_{+\infty}$ and $\mathcal{A}_P$) using a function mirror defined in Fig. 6. The idea behind mirror is simple: if p reflects $P(x)$ then mirror p reflects $P(-x)$. This, among other properties, is expressed below and is proved by induction on p. In §4.3, we use mirror to optimize cooper.

$$
\begin{array}{lcl}
\mathsf{isnorm}_{\mathcal{Z}_+}\ p & \longrightarrow & \mathsf{isnorm}_{\mathcal{Z}_+}\ (\mathsf{mirror}\ p) \wedge (\llparenthesis \mathsf{mirror}\ p \rrparenthesis^{-i \cdot vs} \leftrightarrow \llparenthesis p \rrparenthesis^{i \cdot vs}) \\
 & & \wedge\ \{\llparenthesis t \rrparenthesis_{\tau}^{vs} \mid t \in \{\!\{\mathsf{A}\ p\}\!\}\} = \{-\llparenthesis t \rrparenthesis_{\tau}^{vs} \mid t \in \{\!\{\mathsf{B}\ (\mathsf{mirror}\ p)\}\!\}\}
\end{array}
$$

An implementation The first step in the implementation of cooper is to normalize the formula and to choose the smaller elimination set, possibly mirroring the formula to compensate for this:

$$
\begin{array}{l}
\mathsf{choose}\ p = \\
\textbf{let}\ q = \mathsf{norm}_{\mathcal{Z}_+}\ p;\ (A, B) = (\mathsf{remdups}\ (\mathsf{A}\ q), \mathsf{remdups}\ (\mathsf{B}\ q)) \\
\textbf{in if}\ |B| \leq |A|\ \textbf{then}\ (q, B)\ \textbf{else}\ (\mathsf{mirror}\ q, A)
\end{array}
$$

The main property of choose is that it reduces the $+\infty$ case to the $-\infty$ case:

$$
\begin{array}{l}
\mathsf{qfree}\ p \wedge (\mathsf{choose}\ p = (q, S)) \longrightarrow \\
\mathsf{isnorm}_{\mathcal{Z}_+}\ q \wedge (\llparenthesis \exists\ p \rrparenthesis^{vs} \leftrightarrow \llparenthesis \exists\, q \rrparenthesis^{vs}) \wedge (\{\!\{S\}\!\} = \{\!\{\mathsf{B}\ q\}\!\})
\end{array}
\tag{23}
$$

For (q, S), we generate the right-hand side of (18) and expand the finite quantifiers into disjunctions:

$$
\begin{array}{rl}
\mathsf{explode}(q, S) = & \mathsf{eval}_{\vee}\ (\lambda i. q_{-}[\widehat{i}])\ [1..\mathsf{D}_q] \\
 & \vee\, \mathsf{eval}_{\vee}\ (\lambda t. q[t])\ [t + \widehat{i} \mid t \in S, i \in [1..\mathsf{D}_q]]
\end{array}
$$

The function call $\mathsf{eval}_{\vee}\ f\ [x_1, .., x_n]$ returns the disjunction $f\ x_1 \vee .. \vee f\ x_n$ lazily evaluated: as soon as the disjunct $\boldsymbol{T}$ turns up, the whole expression becomes $\boldsymbol{T}$; $\boldsymbol{F}$ is omitted. The substitution of a term t for $\boldsymbol{v}_0$ in a formula p is performed by $p[t]$. Substitution also simplifies the formula: it evaluates ground terms and relations and performs some logical simplification.

Finally we decrease the de Bruijn indices of the remaining variables using function decr; its definition is obvious and omitted.

The composition of decr and $\mathsf{explode}$ preserves the interpretation:

$$
\begin{array}{l}
\mathsf{isnorm}_{\mathcal{Z}_+}\ p \wedge \{\!\!\{S\}\!\!\} = \{\!\!\{\mathsf{B}\ p\}\!\!\} \rightarrow \\
((\!|\exists\, p|\!)^{vs} \leftrightarrow (\!|\mathsf{decr}(\mathsf{explode}\ (p, B))|\!)^{vs}) \wedge \mathsf{qfree}(\mathsf{decr}(\mathsf{explode}\ (p, B))).
\end{array}
$$

Finally we can define cooper and $\mathsf{qe}_{\mathcal{Z}_+}$:

$$
\begin{aligned}
\mathsf{cooper} &= \mathsf{decr} \circ \mathsf{explode} \circ \mathsf{choose} \\
\mathsf{qe}_{\mathcal{Z}_+} &= \mathsf{qelim\ cooper}
\end{aligned}
$$

Their correctness follows easily:

$$
\begin{aligned}
\mathsf{qfree}\ q \rightarrow \mathsf{qfree}\ (\mathsf{cooper}\ q) \wedge ((\!|\mathsf{cooper}\ q|\!)^{vs} &\leftrightarrow (\!|\exists\, q|\!)^{vs}) \\
\mathsf{qfree}\ (\mathsf{qe}_{\mathcal{Z}_+}\ p) \wedge ((\!|\mathsf{qe}_{\mathcal{Z}_+}\ p|\!)^{vs} &\leftrightarrow (\!|p|\!)^{vs})
\end{aligned}
$$

4.4 Reflecting Ferrante and Rackoff's algorithm

In this section we present a verified implementation of Ferrante and Rackoff's algorithm, which differs only slightly from §4 in [9].

Ferrante and Rackoff's theorem Let U denote $\{(\!|t|\!)^{z\cdot vs}_{\tau}) \mid t \in \{\!\!\{\mathsf{U}\ p\}\!\!\}\}$ and assume $\mathsf{isnorm}_{\mathcal{R}_+} p$. The analogue of Theorem 2 for ϕ-formulae is

$$
(\!|\exists\, p|\!)^{vs} \leftrightarrow (\!|p_- \vee p_+|\!)^{x\cdot vs} \vee \exists (u, u') \in U^2 . (\!|p|\!)^{\frac{u+u'}{2}\cdot vs}. \tag{24}
$$

We prove the (reflected) premises of *fr*, cf. §3.4. Then we have:

$$
\begin{array}{l}
\mathsf{finite}\ \mathsf{U}\ p \\
\exists y. \forall x < y. (\!|p|\!)^{x\cdot vs} \leftrightarrow (\!|p_-|\!)^{x\cdot vs} \\
\exists y. \forall x > y. (\!|p|\!)^{x\cdot vs} \leftrightarrow (\!|p_+|\!)^{x\cdot vs} \\
\neg (\!|p_-|\!)^{x\cdot vs} \wedge (\!|p|\!)^{x\cdot vs} \rightarrow \exists l \in U. l \leq x \\
\neg (\!|p_+|\!)^{x\cdot vs} \wedge (\!|p|\!)^{x\cdot vs} \rightarrow \exists u \in U. x \leq u \\
(\forall y. l < y < u \rightarrow y \notin U) \wedge l < x < u \wedge (\!|p|\!)^{x\cdot vs} \rightarrow \forall y. l < y < u \rightarrow (\!|p|\!)^{y\cdot vs}
\end{array}
$$

All of the above lemmas are proved by induction on p. The proof of the final lemma follows exactly the proof given in §2.4. Using *fr* we obtain (24).

```
$\mathsf{ferrack}\ q = \mathbf{let}\ p = \mathsf{norm}_{\mathcal{R}_+}\ q$
              $U = \mathsf{remdups}(\mathsf{map}\ (\lambda(s,t).\widehat{\tfrac{1}{2}} * (s+t))\ (\mathsf{allpairs}(\mathsf{U}\ p)))$
              $D = \mathsf{eval}_{\vee}\ (\lambda t.p[t])\ U$
           $\mathbf{in}\ \mathsf{decr}(p_- \vee\ p_+ \vee D)$

$\mathsf{qe}_{\mathcal{R}_+}$      $= \mathsf{qelim\ ferrack}$
```

Fig. 7. An implementation

An implementation Figure 7 shows an implementation of ferrack. It first normalizes the input (to p) and computes p_- and p_+ and $\mathsf{U}\ p$ to return the disjunction justified by (24). The de Bruijn indices in the result are decreased via decr. The function call allpairs S returns a list of all the pairs (s,t) such that $(s,t) \in \{\!\{S\}\!\}^2$ or $(t,s) \in \{\!\{S\}\!\}^2$.

The main correctness theorems for ferrack and $\mathsf{qe}_{\mathcal{R}_+}$ are:

$$\begin{aligned} \mathsf{qfree}\ q \longrightarrow \mathsf{qfree}\ (\mathsf{ferrack}\ q) \wedge ((\!|\mathsf{ferrack}\ q|\!)^{vs} \leftrightarrow (\!|\exists\, q|\!)^{vs}) \\ \mathsf{qfree}\ (\mathsf{qe}_{\mathcal{R}_+}\ p) \wedge ((\!|\mathsf{qe}_{\mathcal{R}_+}\ p|\!)^{vs} \leftrightarrow (\!|p|\!)^{vs}) \end{aligned}$$

5 A short comparison of the presented methods

5.1 Implementation

In the tactic-style implementation one does not need to formalize meta-theoretic notions like the syntax of formulae in a particular normal form, e.g. $\mathsf{isnorm}_{\mathcal{Z}_+}$ above. One just goes ahead and writes the tactics. But it is precisely this lack of explicitness that makes tactic writing so tedious and error prone: time and again one finds that tactics fail because the proofs they produce do not fit together as expected, a problem already mentioned in §1. This is where a reflection-based implementation wins: it is developed within the logic and its correctness is proved, i.e. there are no more surprises at runtime. The explicitness of reflection pays off even more during maintenance, where tactics can be awkward to modify.

Due to the progress in sharing theorems with other theorem provers [35, 42], reflected decision procedures are ultimately shared for free. A final advantage of reflection is that it allows to formalize notions like duality, cf. §4.3, which reduces the size of the background theory.

A disadvantage of reflection is that we may need to formalize notions again that are already present on the tactic level. For example, many theorem provers already provide linearization of arithmetic terms as discussed on page 24.

5.2 Efficiency

Efficiency is often considered the key advantage of reflection, since the resulting implementation is exported to ML, where execution is much faster. Coq for instance uses an internal λ-calculus evaluator [27] to perform these computations efficiently.

q	n_q	n_{q0}	n_{q1}	n_{q2}	n_{q3}	$\widehat{c}_{max}$	speedup
1	40	40	0	0	0	24	20
2	27	20	7	0	0	13	50
3	21	2	19	0	0	129	400
4	6	1	0	0	5	6	103
5	5	3	0	5	0	12	50

Fig. 8. Number of quantifiers and speedup in the test formulae for $\mathcal{Z}_+$

q	n_q	n_{q0}	n_{q1}	n_{q2}	n_{q3}	$\widehat{c}_{max}$	speedup
2	10	5	5	0	0	500	43
3	15	5	5	5	0	100	44
4	20	5	5	5	5	1200	64

Fig. 9. Number of quantifiers and speedup in the test formulae for $\mathcal{R}_+$

Our implementations in Isabelle/HOL confirms the efficiency advantage: on average, the reflection-based decision procedure for $\mathcal{Z}_+$ is 130 times faster that the tactic-based one, for $\mathcal{R}_+$ the speedup is still 60. The formulae we tested are either from the literature or encoutered subgoals in Isabelle theories. The details of our measurements are shown in figures 8 and 9. Each line describes a sample set. Column 1, q, gives the number of quantifiers in each sample formula. The formulae contain up to five quantifiers and three quantifier alternations. The number n_q represents the number of formulae with q quantifiers. We have $n_q = n_{q0}+n_{q1}+n_{q2}+n_{q3}$, where n_{qi} is the number of formulae containing q quantifiers and i quantifier alternations. The column $\widehat{c}_{max}$ gives the maximal constant occurring in the given set of formulae. The last column gives the speedup. The tactic style methods presented in §3.3 and §3.4 needed 463 and 378 seconds, to solve all the problems by inference. The ML implementations obtained by Isabelle's code generator from the formally verified procedures presented in §4.3 and in §4.4 took 3.48 and 5.96 seconds, a speedup of 133 and 63. All timings were carried out on a PowerBook G4 with a 1.67 GHz processor running OSX.

Acknowledgement

We thank John Harrison and Michael Norrish for useful discussions and comments.

References

1. H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *J. Autom. Reasoning*, 28(3):321–336, 2002.
2. S. Berghofer and T. Nipkow. Executing higher order logic. In *In Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lect. Notes in Comp. Sci.*, pages 24–40. Springer-Verlag, 2002.
3. L. Berman. Precise bounds for Presburger arithmetic and the reals with addition: Preliminary report. In *FOCS*, pages 95–99. IEEE, 1977.
4. L. Berman. The complexitiy of logical theories. *Theoretical Compututer Science*, 11:71–77, 1980.
5. Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Text in theor. comp. science: an EATCS series.* Springer, 2004.
6. B. Boigelot, S. Jodogne, and P. Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Trans. Comput. Log.*, 6(3):614–633, 2005.
7. R. S. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–84. Academic Press, New York, 1981.
8. A. Chaieb. Isabelle trifft Presburger Arithmetik. Master's thesis, TU München, 2003.
9. A. Chaieb. Verifying mixed real-integer quantifier elimination. In Furbach and Shankar [22], pages 528–540.
10. A. Chaieb and T. Nipkow. Verifying and reflecting quantifier elimination for Presburger arithmetic. In G. Stutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2005.
11. Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
12. George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.
13. D.C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.
14. Pierre Crégut. Une procédure de décision réflexive pour un fragment de l'arithmétique de Presburger. In *Informal proceedings of the 15th journées francophones des langages applicatifs*, 2004.
15. Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.
16. M. Davis. A computer program for Presburger's algorithm. In *Summaries of talks presented at the Summer Inst. for Symbolic Logic, Cornell University*, pages 215–233. Inst. for Defense Analyses, Princeton, NJ, 1957.

17. L. Dines. Systems of linear inequalities. *Annals of Math.*, 20:191–199, 1919.
18. Herbert Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.
19. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.*, 4(1):69–76, 1975.
20. M. Fischer and M. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAMAMS: Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, 1974.
21. J. Fourier. Solution d'une question particulière du calcul des inégalités. *Nouveau Bulletin des Sciences par la Société Philomatique de Paris*, pages 99–100, 1823.
22. Ulrich Furbach and Natarajan Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lect. Notes in Comp. Sci.* Springer, 2006.
23. M. Fürer. The complexity of Presburger arithmetic with bounded quantifier alternation depth. *Theor. Comput. Sci.*, 18:105–111, 1982.
24. M.C.J. Gordon, Robin Milner, and C.P. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1979.
25. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher order logic.* Cambridge University Press, 1993.
26. E. Grädel. Subclasses of Presburger arithmetic and the polynomial-time hierarchy. *Theor. Comput. Sci.*, 56:289–301, 1988.
27. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Int. Conf. Functional Programming*, pages 235–246. ACM Press, 2002.
28. J. Harrison. *HOL Light Tutorial (for version 2.20)*, September 2006.
29. John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz`.
30. John Harrison. *Theorem proving with the real numbers.* PhD thesis, University of Cambridge, Computer Laboratory, 1996.
31. F. Klaedtke. On the automata size for Presburger arithmetic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 110–119. IEEE Computer Society Press, 2004.
32. R. Klapper and A. Stump. Validated Proof-Producing Decision Procedures. In C. Tinelli and S. Ranise, editors, *2nd Int. Workshop Pragmatics of Decision Procedures in Automated Reasoning*, 2004.
33. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.
34. Assia Mahboubi. *Contributions à la certification des calculs sur $\mathbb{R}$: théorie, preuves,programmation.* PhD thesis, Université de Nice Sophia-Antipolis, 2006.
35. S. McLaughlin. An interpretation of isabelle/hol in hol light. In Furbach and Shankar [22], pages 192–204.
36. S. McLaughlin and J. Harrison. A Proof-Producing Decision Procedure for Real Arithmetic. In Robert Nieuwenhuis, editor, *CADE-20: 20th International Conference on Automated Deduction, proceedings*, volume 3632 of *Lect. Notes in Comp. Sci.*, pages 295–314. Springer-Verlag, 2005.
37. T. S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen.* PhD thesis, Universität Zürich, 1936.
38. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Palo Alto Research center, 1981.

39. T. Nipkow. Functional unification of higher-order patterns. In *8th IEEE Symp. Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.
40. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. `http://www.in.tum.de/~nipkow/LNCS2283/`.
41. M. Norrish. Complete integer decision procedures as derived rules in HOL. In D.A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2003*, volume 2758 of *Lect. Notes in Comp. Sci.*, pages 71–86. Springer-Verlag, 2003.
42. S. Obua and S. Skalberg. Importing hol into isabelle/hol. In Furbach and Shankar [22], pages 298–302.
43. D. C. Oppen. Elementary bounds for presburger arithmetic. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 34–37, New York, NY, USA, 1973. ACM Press.
44. L. C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.
45. M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I Congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.
46. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.
47. C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 320–325, New York, NY, USA, 1978. ACM Press.
48. B. Scarpellini. Complexity of subclasses of Presburger arithmetic. *Trans. AMS*, 284:203–218, 1984.
49. T. Skolem. Über einige Satzfunktionen in der Arithmetik. In *Skrifter utgitt av Det Norske Videnskaps-Akademi i Oslo, I. Matematisk naturvidenskapelig klasse*, volume 7, pages 1–28. , Oslo, 1931.
50. A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 2d edition, 1951.
51. V. Weispfenning. The complexity of linear problems in fields. *J. Symb. Comput.*, 5(1/2):3–27, 1988.
52. V. Weispfenning. The complexity of almost linear diophantine problems. *J. Symb. Comput.*, 10(5):395–404, 1990.
53. V. Weispfenning. Complexity and uniformity of elimination in Presburger arithmetic. In *ISSAC*, pages 48–53, 1997.
54. V. Weispfenning. Mixed real-integer linear quantifier elimination. In *ISSAC '99: Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, pages 129–136, New York, NY, USA, 1999. ACM Press.
55. P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *SAS '95: Proc. of the Second Int. Symp. on Static Analysis*, pages 21–32, London, UK, 1995. Springer-Verlag.