Linear Quantifier Elimination

Tobias Nipkow

Institut für Informatik, Technische Universität München

Abstract. This paper presents verified quantifier elimination procedures for dense linear orders (DLO), for real and for integer linear arithmetic. The DLO procedures are new. All procedures are defined and verified in the theorem prover Isabelle/HOL, are executable and can be applied to HOL formulae themselves (by reflection).

1 Introduction

This paper is about the concise implementation of *quantifier elimination* (QE) procedures (QEPs) for linear arithmetics. QE is a venerable logical technique which yields decision procedures if ground atoms are decidable. The focus of our work is the compact implementation of QEPs (for linear arithmetics) inside a theorem prover. All our QEPs have been defined and verified in Isabelle/HOL [16]. We do not discuss these formal proofs here. They are detailed, mostly structured and available online at afp.sf.net, together with the QEPs themselves. Because the informal proofs of these QEPs can be found in the literature, they need not be discussed either. The exception are our two new QEPs for which informal correctness proofs are given.

The main contributions of this paper are:

- Two new QEPs for dense linear orders (DLO) inspired by QEPs for linear real arithmetic.
- Presentation of 5 verified implementations of QEPs: two for DLO, two for linear real arithmetic and one for Presburger arithmetic (Cooper). We show everything but the most trivial details, providing reference implementations and convincing the reader that nothing has been swept under the carpet.
- Extremely compact formalizations due to the almost excessive use of lists and list comprehensions.
- A common reusable QE framework using Isabelle's structuring facility of locales, thus factoring out the common parts of the different QEPs.

Why this obsession with executable and verified QEPs? The context of this research is the question of how to implement trustworthy and efficient decision procedures in foundational theorem provers, i.e. without having to trust an external oracle. *Reflection*, originally proposed by Boyer and Moore [2] and used to great effect in systems like Coq (e.g. [7]) and Isabelle (e.g. [4]) has become a standard approach. Suffice it to say that we follow this approach, too, and that all the algorithms in this paper can be used directly on formulae in Isabelle — details can be found elsewhere (e.g. [15]).

This paper is a contribution to the growing body of verified theorem proving algorithms. In spirit it is close to Harrison's forthcoming book [9] which presents all algorithms in OCaml. Only that our code is verified.

It should be emphasized that the presentation is streamlined for succinctness. In particular, we always restrict attention to two of the four relations $=$, $<$, $\leq$, $\neq$. For example, in DLOs it suffices to consider $=$ and $<$ because $x \leq y$ is equivalent with $x < y \vee x = y$ and $x \neq y$ is equivalent with $x < y \vee y < x$. For QEPs based on DNF this is a disaster because it leads to further case splits. The algorithms in this paper avoid DNF. Nevertheless, an efficient implementation would always work with all four relations. The corresponding generalization of our code is straightforward.

The paper is structured as follows. In §3 we describe a HOL model of logical formulae parameterized by a language of atoms and present a generic QEP parameterized by a QEP for a single quantifier. The remaining sections present a succession of 5 single-quantifier QEPs for different linear theories.

2 Basic Notation

HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

The types of truth values, natural numbers, integers and reals are called *bool*, *nat*, *int* and *real*. The space of total functions is denoted by $\Rightarrow$. Type variables are denoted by α, β, etc. The notation $t{::}\tau$ means that term t has type τ.

Sets over type α, type α *set*, follow the usual mathematical convention.

Lists over type α, type α *list*, come with the empty list $[]$, the infix constructor $\cdot$, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in s usually stand for lists. In addition to the standard functions *map* and *filter*, Isabelle/HOL also supports Haskell-style list comprehension notation, with minor differences: instead of `[e | x <- xs, ...]` we write $[e.\ x \leftarrow xs, \ldots]$, and $[x{\leftarrow}xs.\ \ldots]$ is short for $[x.\ x{\leftarrow}xs, \ldots]$.

Finally note that $=$ on type *bool* means "iff".

During informal explanations we often switch to everyday mathematical notation where (a, b) can be a pair or an open interval.

3 Logic

Formulae are defined as a recursive datatype with a parameter type α of atoms:

$$\begin{aligned}\textbf{datatype}\ \alpha\ fm = {} & \top \mid \bot \mid A\ \alpha \\ & \mid (\alpha\ fm) \wedge (\alpha\ fm) \mid (\alpha\ fm) \vee (\alpha\ fm) \mid \neg\ (\alpha\ fm) \mid \exists\ (\alpha\ fm)\end{aligned}$$

The **boldface** symbols $\wedge$, $\vee$, $\neg$ and $\exists$ are ordinary constructors chosen to resemble the logical operators they represent. Constructor A encloses atoms. The type of atoms is left open by making it a parameter α. Variables are represented by de

Bruijn indices: quantifiers do not explicitly mention the name of the variable being bound because that is implicit. For example, $\exists$ ($\exists$ ... *0* ... *1* ...) represents a formula $\exists x_1.\exists x_0.\ \ldots x_0 \ldots x_1 \ldots$. Note that the only place where variables can appear is inside atoms. The only distinction between free and bound variables is that the index of a free variable is larger than the number of enclosing binders.

3.1 Auxiliary Functions

The constructors $\vee$, $\wedge$ and $\neg$ have optimized ("short-circuit") versions *or*, *and* and *neg*: $or\ \top\ \varphi = \top$, $or\ \varphi\ \top = \top$, $or\ \bot\ \varphi = \varphi$, $or\ \varphi\ \bot = \varphi$ and $or\ \varphi_1\ \varphi_2 = (\varphi_1 \vee \varphi_2)$ otherwise; $and\ \top\ \varphi = \varphi$, $and\ \varphi\ \top = \varphi$, $and\ \bot\ \varphi = \bot$, $and\ \varphi\ \bot = \bot$ and $and\ \varphi_1\ \varphi_2 = (\varphi_1 \wedge \varphi_2)$ otherwise; $neg\ \top = \bot$, $neg\ \bot = \top$ and $neg\ \varphi = \neg\ \varphi$ otherwise.

Disjunction of a lists of formulae is easily defined:

$$list\text{-}disj\ [\varphi_1,\ldots,\varphi_n] = or\ \varphi_1\ (or\ \ldots\ \varphi_n)$$

Most of our work will be concerned with quantifier-free formulae where all negations have not just been pushed right in front of atoms but actually into them. This is easy for linear orders because $\neg(x < y)$ is equivalent with $y \leq x$. This conversion will be described later on because it depends on the type of atoms. The (trivial to define) predicates

$$qfree,\ nqfree :: \alpha\ fm \Rightarrow bool$$

check whether their argument is free of quantifiers (*qfree*), and free of negations and quantifiers (*nqfree*).

There are also two mapping functionals

$$\begin{aligned} map_{fm} &:: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ fm \Rightarrow \beta\ fm \\ amap_{fm} &:: (\alpha \Rightarrow \beta\ fm) \Rightarrow \alpha\ fm \Rightarrow \beta\ fm \end{aligned}$$

where $map_{fm}\ f$ is the canonical one that simply replaces $A\ a$ by $A\ (f\ a)$, whereas $amap_{fm}$ may also simplify the formula via *and*, *or* and *neg*:

$$\begin{aligned} &amap_{fm}\ h\ \top = \top \qquad amap_{fm}\ h\ \bot = \bot \qquad amap_{fm}\ h\ (A\ a) = h\ a \\ &amap_{fm}\ h\ (\varphi_1 \wedge \varphi_2) = and\ (amap_{fm}\ h\ \varphi_1)\ (amap_{fm}\ h\ \varphi_2) \\ &amap_{fm}\ h\ (\varphi_1 \vee \varphi_2) = or\ (amap_{fm}\ h\ \varphi_1)\ (amap_{fm}\ h\ \varphi_2) \\ &amap_{fm}\ h\ (\neg\ \varphi) = neg\ (amap_{fm}\ h\ \varphi) \end{aligned}$$

Both mapping functionals are only defined and needed for *qfree* formulae.

The set of atoms in a formula is computed by the (trivial to define) function $atoms :: \alpha\ fm \Rightarrow \alpha\ set$.

3.2 Interpretation

The interpretation or semantics of a *fm* is defined via the obvious homomorphic mapping to an HOL formula: $\wedge$ becomes $\wedge$, $\vee$ becomes $\vee$, etc. The interpretation

of atoms is a parameter of this mapping. Atoms may refer to variables and are thus interpreted w.r.t. a valuation. Since variables are represented as natural numbers, the valuation is naturally represented as a list: variable i refers to the ith entry in the list (starting with 0). This leads to the following interpretation function $interpret :: (\alpha \Rightarrow \beta\ list \Rightarrow bool) \Rightarrow \alpha\ fm \Rightarrow \beta\ list \Rightarrow bool$:

$interpret\ h\ \top\ xs = True \qquad interpret\ h\ \bot\ xs = False$
$interpret\ h\ (A\ a)\ xs = h\ a\ xs$
$interpret\ h\ (\varphi_1 \wedge \varphi_2)\ xs = (interpret\ h\ \varphi_1\ xs \wedge interpret\ h\ \varphi_2\ xs)$
$interpret\ h\ (\varphi_1 \vee \varphi_2)\ xs = (interpret\ h\ \varphi_1\ xs \vee interpret\ h\ \varphi_2\ xs)$
$interpret\ h\ (\neg\ \varphi)\ xs = (\neg\ interpret\ h\ \varphi\ xs)$
$interpret\ h\ (\exists\ \varphi)\ xs = (\exists\, x.\ interpret\ h\ \varphi\ (x \cdot xs))$

In the equation for $\exists$ the value of the bound variable x is added at the front of the valuation. De Bruijn indexing ensures that in the body 0 refers to x and $i+1$ refers to bound variable i further up.

3.3 Atoms

Atoms are more than a type parameter α. They come with an *interpretation* (their semantics), and a few other specific functions. These functions are also parameters of the generic part of quantifier elimination. Thus the further development will be like a module parameterized with the type of atoms and some functions on atoms. These parameters will be instantiated later on when applying the framework to various linear arithmetics.

In Isabelle this parameterization is achieved by means of a **locale** [1], a named context of types, functions and assumptions about them. We call this context *ATOM*. It provides the following functions

$$\begin{array}{ll} I_a & :: \alpha \Rightarrow \beta\ list \Rightarrow bool \\ aneg & :: \alpha \Rightarrow \alpha\ fm \\ depends_0 & :: \alpha \Rightarrow bool \\ decr & :: \alpha \Rightarrow \alpha \end{array}$$

with the following intended meaning:

$I_a\ a\ xs$ is the interpretation of atom a w.r.t. valuation xs, where variable i (note $i :: nat$ because of de Bruijn) is assigned the ith element of xs.

aneg negates an atom. It returns a formula which should be free of negations. This is strictly for convenience: it means we can eliminate all negations from a formula. In the worst case we would have to introduce negated versions of all atoms, but in the case of linear orders this is not necessary because we can turn, for example, $\neg(x < y)$ into $(y < x) \vee (y = x)$.

$depends_0\ a$ checks if atom a contains (depends on) variable 0 and *decr* a decrements every variable in a by 1.

Within context *ATOM* we introduce the abbreviation $I \equiv interpret\ I_a$. The assumptions on the parameters of *ATOM* can now be stated quite succinctly:

$$I\ (aneg\ a)\ xs = (\neg\ I_a\ a\ xs) \qquad nqfree\ (aneg\ a)$$
$$\neg\ depends_0\ a \Longrightarrow I_a\ a\ (x \cdot xs) = I_a\ (decr\ a)\ xs$$

Function *aneg* must return a quantifier and negation-free formula whose interpretation is the negation of the input. And when interpreting an atom not containing variable 0 we can drop the head of the valuation and decrement the variables without changing the interpretation.

These assumptions must be discharged when the locale is instantiated. We do not show this in the text because the proofs are straightforward in all cases.

In the context of *ATOM* we define two auxiliary functions: $atoms_0\ \varphi$ computes the list of all atoms in φ that depend on variable 0. The *negation normal form* (NNF) of a *qfree* formula is defined in the customary manner by pushing negations inwards. We show only a few representative equations:

$$nnf\ (\neg\ (A\ a)) = aneg\ a$$
$$nnf\ (\varphi_1 \vee \varphi_2) = (nnf\ \varphi_1 \vee nnf\ \varphi_2)$$
$$nnf\ (\neg\ (\varphi_1 \vee \varphi_2)) = (nnf\ (\neg\ \varphi_1) \wedge nnf\ (\neg\ \varphi_2))$$
$$nnf\ (\neg\ (\varphi_1 \wedge \varphi_2)) = (nnf\ (\neg\ \varphi_1) \vee nnf\ (\neg\ \varphi_2))$$

The first equation differs from the usual definition and gets rid of negations altogether — see the explanation of *aneg* above.

3.4 Quantifier Elimination

The elimination of all quantifiers from a formula is achieved by eliminating them one by one in a bottom-up fashion. Thus each step needs to deal merely with the elimination of a single quantifier in front of a quantifier-free formula. This step is theory-dependent and hard. The lifting to arbitrary formulae is simple and can be done once and for all. We assume we are given a function $qe :: \alpha\ fm \Rightarrow \alpha\ fm$ for the elimination of a single $\exists$, i.e. $I\ (qe\ \varphi) = I\ (\exists\ \varphi)$ if *qfree* φ. Note that *qe* is not applied to $\exists\ \varphi$ but just to φ, $\exists$ remains implicit. Lifting *qe* is straightforward:

$$lift\text{-}nnf\text{-}qe :: (\alpha\ fm \Rightarrow \alpha\ fm) \Rightarrow \alpha\ fm \Rightarrow \alpha\ fm$$

$$lift\text{-}nnf\text{-}qe\ qe\ (\varphi_1 \wedge \varphi_2) = and\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_1)\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_2)$$
$$lift\text{-}nnf\text{-}qe\ qe\ (\varphi_1 \vee \varphi_2) = or\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_1)\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi_2)$$
$$lift\text{-}nnf\text{-}qe\ qe\ (\neg\ \varphi) = neg\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)$$
$$lift\text{-}nnf\text{-}qe\ qe\ (\exists\ \varphi) = qe\ (nnf\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi))$$
$$lift\text{-}nnf\text{-}qe\ qe\ \varphi = \varphi$$

Note that *qe* is called with an argument already in NNF. We can go even further and put the argument of *qe* into DNF. This is detailed elsewhere [15] but avoided here because it can lead to non-elementary complexity.

3.5 Correctness

Correctness *lift-nnf-qe* is roughly expressed as follows: if *qe* eliminates one existential while preserving the interpretation, then *lift-nnf-qe qe* eliminates all quantifiers while preserving the interpretation.

For compactness we employ a set theoretic language for expressing properties of functions: $A \to B$ is the set of functions from A to B and $|P| \equiv \{x \mid P\ x\}$.

Elimination of all quantifiers is easy:

Lemma 1. *If* $qe \in |nqfree| \to |qfree|$ *then* $qfree\ (\mathit{lift\text{-}nnf\text{-}qe}\ qe\ \varphi)$.

Preservation of the interpretation is slightly more involved:

Lemma 2. *If* $qe \in |nqfree| \to |qfree|$ *and for all* φ *and* xs: $(nqfree\ \varphi \Longrightarrow I\ (qe\ \varphi)\ xs = (\exists\, x.\ I\ \varphi\ (x \cdot xs)))$, *then* $I\ (\mathit{lift\text{-}nnf\text{-}qe}\ qe\ \varphi)\ xs = I\ \varphi\ xs$.

In the following sections we define a number of quantifier elimination functions called f_1 (for different names f) that eliminate a single $\exists$. In each case we have proved that f_1 satisfies the assumptions of the above two lemmas (with f_1 for qe), define $f = \mathit{lift\text{-}nnf\text{-}qe}\ f_1$ and thus obtain $qfree\ (f\ \varphi)$ and $I\ (f\ \varphi)\ xs = I\ \varphi\ xs$ as corollaries. Because of this uniformity and because the correctness proofs are either discussed informally beforehand or are well-known from the literature, we suppress all of this in the presentation. Thus it may look as if we merely present code, but the proofs are all there!

4 Dense Linear Orders

The theory of dense linear orders (without endpoints) is an extension of the theory of linear orders with the axioms

$$x < z \Longrightarrow \exists\, y.\ x < y \wedge y < z \qquad \exists\, u.\ x < u \qquad \exists\, l.\ l < x$$

It is the canonical example of quantifier elimination [11]. The equivalence $(\exists\, y.\ x < y \wedge y < z) = (x < z)$ is an easy consequence of the axioms and the essence of Fourier's elimination method, which requires conversion to DNF and is thus non-elementary.

In contrast we develop two new NNF-based algorithms based on the *test point* method (originally due to Cooper [5] and Ferrante and Rackoff [6] and later generalized by Weispfenning [19]). The idea is to find a finite set of test points T (depending on φ) such that $(\exists x.\ \varphi(x)) = (\bigvee_{t \in T} \varphi(t))$. The complication is that (conceptually) T may contain values like infinity, infinitesimals or intermediate points, values that are not representable in the given term language. The challenge is to define special versions of substitution for these values.

4.1 Atoms

There are just the two relations $<$ and $=$ and no function symbols. Thus atomic formulae can be represented by the following datatype:

$$\textbf{datatype}\ \mathit{atom} = \mathit{nat} \mathbin{\boldsymbol{<}} \mathit{nat} \mid \mathit{nat} \mathbin{\boldsymbol{=}} \mathit{nat}$$

Note the **bold** infix constructors $\boldsymbol{<}$ and $\boldsymbol{=}$. Because there are no function symbols, the arguments of the relations must be variables. For example, $i \mathbin{\boldsymbol{<}} j$ represents the atom $x_i < x_j$ in de Bruijn notation.

Now we can instantiate locale *ATOM*. Type parameter α becomes type *atom*. The interpretation function I_a becomes I_{dlo} where

$$I_{dlo}\ (i = j)\ xs = (xs_{[i]} = xs_{[j]}) \qquad I_{dlo}\ (i < j)\ xs = (xs_{[i]} < xs_{[j]})$$

The notation $xs_{[i]}$ means selection of the ith element of xs. The type of I_{dlo} is explicitly restricted such that xs must be a list of elements over a dense linear order, where the latter is formalized as a type class [8] with the axioms shown at the start of this section. Thus all valuations in this section are over dense linear orders. Parameter *aneg* becomes neg_{dlo}:

$$neg_{dlo}\ (i < j) = (A\ (j < i) \vee A\ (i = j))$$
$$neg_{dlo}\ (i = j) = (A\ (i < j) \vee A\ (j < i))$$

The parameters *adepends* and *adecr* are instantiated with $depends_{dlo}$ and $decr_{dlo}$:

$$depends_{dlo}\ (i = j) = (i = 0 \vee j = 0)$$
$$depends_{dlo}\ (i < j) = (i = 0 \vee j = 0)$$

$$decr_{dlo}\ (i < j) = (i - 1 < j - 1) \qquad decr_{dlo}\ (i = j) = (i - 1 = j - 1)$$

This instantiation satisfies all the axioms of *ATOM*.

4.2 The Interior Point Method

Ferrante and Rackoff [6] realized (for linear real arithmetic) that when eliminating x from ϕ it (essentially) suffices to collect all lower bounds l of x (i.e. $l < x$ occurs in ϕ) and all upper bounds u of x (i.e. $x < u$ occurs in ϕ) and try all such $(l + u)/2$ as test points. This method is implemented in §5.2.

Now we present a novel quantifier elimination method for DLO based on Ferrante and Rackoff's idea. The problem with DLO is that one cannot name any point between two variables x and y. Hence a special form of substitution must be defined that behaves as if some intermediate point was substituted without requiring such a point. We use the symbolic notation $x{\downarrow}y$ to denote some arbitrary but fixed point in the interval (x, y). The key cases in defining substitution with $x{\downarrow}y$ are: $(x{\downarrow}y < z) = (y \leq z)$, $(z < x{\downarrow}y) = (z \leq x)$, $(x{\downarrow}y < x{\downarrow}y) = False$, $(x{\downarrow}y = x{\downarrow}y) = True$ and $(x{\downarrow}y = z) = False$. The last equation is motivated because we can always choose $x{\downarrow}y$ to be different from z. Note also that these definitions only work as expected if $x < y$.

We also need the fictitious values $-\infty$ and ∞ first used by Cooper. Then we can formulate the interior point method as a logical equivalence in test point form, where ϕ must be quantifier-free and in NNF:

$$(\exists x.\ \phi(x)) \ = \ (\phi(-\infty) \vee \phi(\infty) \vee \bigvee_{y \in E} \phi(y) \vee \bigvee_{y \in L, z \in U} (y < z \wedge \phi(y{\downarrow}z))) \qquad (1)$$

E is the set of y such that $x = y$ or $y = x$ occur in $\phi(x)$, L is the set of y such that $y < x$ occurs in $\phi(x)$, U is the set of y such that $x < y$ occurs in $\phi(x)$, where x is the bound variable and y is different from x.

We sketch a proof of (1), details can be found in the Isabelle proof. The if-direction is easy as in each case a witness is given. Except that $-\infty$, ∞ and $y{\downarrow}z$ are not proper values. But by induction on ϕ one can show that $\phi(-\infty)$ etc imply $\phi(x)$ for suitable x:

$$\exists x.\forall y \le x.\ \phi(-\infty) = \phi(y) \qquad \exists x.\forall y \ge x.\ \phi(\infty) = \phi(y)$$
$$y < z \wedge \phi(y{\downarrow}z) \Longrightarrow \forall x \in (y,z).\ \phi(x)$$

For the only-if-direction assume $\phi(x)$ and not $\phi(-\infty) \vee \phi(\infty) \vee \bigvee_{y\in E} \phi(y)$. We have to show that $\phi(y{\downarrow}z)$ for some $y \in L$ and $z \in U$. From the assumptions it follows by induction on ϕ that there must be $y_0 \in L$ and $z_0 \in U$ such that $x \in (y_0, z_0)$. Now we show (by induction on ϕ) the lemma that innermost intervals (y, z) completely satisfy ϕ:

Lemma 3. *If $x \in (y,z)$, $x \notin E$, $(y,x) \cap L = \emptyset$ and $(x,z) \cap U = \emptyset$, then $\phi(x)$ implies $\forall u \in (y,z).\ \phi(u)$.*

Given $x \in (y_0, z_0)$ we define $y = \max\{y \in L \mid y < x\}$ and $z = \min\{z \in U \mid x < z\}$. It is easy to see that this satisfies the premises of the lemma and hence $\forall u \in (y,z).\ \phi(u)$. Again by induction on ϕ one can show that this actually implies $\phi(y{\downarrow}z)$:

Lemma 4. *If $x \in (y,z)$, $x \notin E$, $(y,x) \cap L = \emptyset$ and $(x,z) \cap U = \emptyset$, then $(\forall x \in (y,z).\ \phi(x))$ implies $\phi(y{\downarrow}z)$.*

4.3 A Verified Implementation of the Interior Point Method

The executable version of (1) is short but requires some auxiliary functions.

```
interior₁ φ =
(let as = atoms₀ φ; lbs = lbounds as; ubs = ubounds as; ebs = ebounds as;
     intrs = [ A(l < u) ∧ (subst₂ l u φ). l←lbs, u←ubs]
 in list-disj (inf₋ φ · inf₊ φ · intrs @ map (subst φ) ebs))
```

We will now explain the ingredients.

The implementation of substituting $l{\downarrow}u$ in atoms is given below. Please note that substitution must not just substitute for variable 0 but must also decrement the other variables.

$$asubst_2\ l\ u\ (0 < 0) = \bot \qquad asubst_2\ l\ u\ (Suc\ i < Suc\ j) = A\ (i < j)$$
$$asubst_2\ l\ u\ (0 < Suc\ j) = (A\ (u < j) \vee A\ (u = j))$$
$$asubst_2\ l\ u\ (Suc\ i < 0) = (A\ (i < l) \vee A\ (i = l))$$
$$asubst_2\ l\ u\ (0 = 0) = \top \qquad asubst_2\ l\ u\ (Suc\ i = Suc\ j) = A\ (i = j)$$
$$asubst_2\ l\ u\ (0 = Suc\ v) = \bot \qquad asubst_2\ l\ u\ (Suc\ v = 0) = \bot$$

From atoms to formulae is a short step: $subst_2\ l\ u\ \varphi \equiv amap_{fm}\ (asubst_2\ l\ u)\ \varphi$

Plain old substitution of one variable for 0 is defined first on variables, then on atoms and finally on formulae:

$$isubst\ k\ 0 = k \qquad isubst\ k\ (Suc\ i) = i$$

$asubst\ k\ (i < j) = (isubst\ k\ i < isubst\ k\ j)$
$asubst\ k\ (i = j) = (isubst\ k\ i = isubst\ k\ j)$

$subst\ \varphi\ k \equiv map_{fm}\ (asubst\ k)\ \varphi$

Substituting $-\infty$ for 0 is implemented as follows:

$amin\text{-}inf\ (i < 0) = \bot$ $\quad$ $amin\text{-}inf\ (0 < Suc\ j) = \top$
$amin\text{-}inf\ (Suc\ i < Suc\ j) = A\ (i < j)$
$amin\text{-}inf\ (0 = 0) = \top$ $\quad$ $amin\text{-}inf\ (Suc\ i = Suc\ j) = A\ (i = j)$
$amin\text{-}inf\ (0 = Suc\ v) = \bot$ $\quad$ $amin\text{-}inf\ (Suc\ v = 0) = \bot$
$inf_{-}\ \varphi \equiv amap_{fm}\ amin\text{-}inf\ \varphi$

Dually there is inf_{+} for substituting ∞. Lower bounds, upper bounds and equalities are conveniently collected from a list of atoms by list comprehension:

$lbounds\ as = [i.\ (Suc\ i < 0) \leftarrow as]$ $\quad$ $ubounds\ as = [i.\ (0 < Suc\ i) \leftarrow as]$
$ebounds\ as = [i.\ (Suc\ i = 0) \leftarrow as]\ @\ [i.\ (0 = Suc\ i) \leftarrow as]$

4.4 The Method of Infinitesimals

Loos and Weispfenning [12] proposed a quantifier elimination procedure for linear real arithmetic (see §5.3) where test points are $x + \varepsilon$ (for x a lower bound) or $y - \varepsilon$ (for y an upper bound) where ε is an infinitesimal. That is, the test points are arbitrarily close to the lower or upper bounds of the eliminated variable. In particular, it is not necessary to pair all lower and upper bounds but one can choose either set, typically the smaller one. For succinctness we ignore this duality and concentrate on the lower bounds only.

In this section we adapt the idea of infinitesimals to derive a new quantifier elimination procedure for DLO. We merely need to explain what substitution of $x+\varepsilon$ means: $(x+\varepsilon < y) = (x < y)$, $(y < x+\varepsilon) = (y \le x)$, $(x+\varepsilon < x+\varepsilon) = False$, $(x+\varepsilon = x+\varepsilon) = True$, $(x+\varepsilon = y) = False$, where x and y are different variables.

The test point method with infinitesimals is justified by the following equivalence, where, as usual, ϕ is quantifier free and in NNF:

$$(\exists x.\ \phi(x)) \ = \ (\phi(-\infty) \vee \bigvee_{y \in E} \phi(y) \vee \bigvee_{y \in L} \phi(y + \varepsilon)) \tag{2}$$

where E and L are defined as in (1). The proof is also similar. The main differences are: For the if-direction we need to show (by induction on ϕ) that $y + \varepsilon$ represents a proper witness:

$$\phi(y + \varepsilon) \Longrightarrow \exists y' > y. \forall x \in (y, y').\ \phi(x)$$

The two lemmas for the only-if-direction become

Lemma 5. *If $y < x$, $x \notin E$, $(y, x) \cap L = \emptyset$ and $\phi(x)$, then $\forall u \in (y, x].\ \phi(u)$.*

Lemma 6. *If $y < x$, $x \notin E$, $(y, x) \cap L = \emptyset$ and $\forall u \in (y, x].\ \phi(u)$, then $\phi(y + \varepsilon)$.*

Our verified implementation of (2)

$$eps_1\ \varphi = (\mathit{let}\ as = atoms_0\ \varphi;\ lbs = lbounds\ as;\ ebs = ebounds\ as \\ \mathit{in}\ list\text{-}disj\ (inf_-\ \varphi\ \cdot\ map\ (subst_+\ \varphi)\ lbs\ @\ map\ (subst\ \varphi)\ ebs))$$

requires only one new concept, $subst_+\ \varphi\ y$, the substitution $\phi(y + \varepsilon)$:

$$asubst_+\ k\ (0 < 0) = \bot \qquad asubst_+\ k\ (Suc\ i < Suc\ j) = A\ (i < j)$$
$$asubst_+\ k\ (0 < Suc\ j) = A\ (k < j)$$
$$asubst_+\ k\ (Suc\ i < 0) = (\mathit{if}\ i = k\ \mathit{then}\ \top\ \mathit{else}\ A\ (i < k) \vee A\ (i = k))$$
$$asubst_+\ k\ (0 = 0) = \top \qquad asubst_+\ k\ (Suc\ i = Suc\ j) = A\ (i = j)$$
$$asubst_+\ k\ (0 = Suc\ v) = \bot \qquad asubst_+\ k\ (Suc\ v = 0) = \bot$$
$$subst_+\ \varphi\ k \equiv amap_{fm}\ (asubst_+\ k)\ \varphi$$

4.5 Complexity

A formula of size n can contain at most n variables. The set of variables decreases by one in each step. In the worst case all of them are bound and need to be eliminated. In each step of the quantifier elimination processes (1) and (2) the sets E, L and U are at most as large as k, the current number of variables.

The interior point method makes at most $(k-1)^2$ copies of the formula in each step. Hence the size of the output formula and also the amount of working space required is $O(n \cdot (n-1)^2 \cdots 1^2) = O(n \cdot (n-1)!^2)$. The method of infinitesimals, however, only makes at most $k-1$ copies, thus requiring only $O(n \cdot (n-1) \cdots 1) = O(n!)$ space. The time complexity of both algorithms is linear in their space complexity, i.e. time and space coincide.

5 Linear Real Arithmetic

Linear real arithmetic is concerned with terms built up from variables, constants, addition, and multiplication with constants. Relations between such terms can be put into a normal form $r \bowtie c_0 * x_0 + \cdots c_n * x_n$ with $\bowtie\ \in \{=, <\}$ and $r, c_0, \ldots, c_n \in \mathbb{R}$. It is this normal form we work with in this section.

Note that although we phrase everything in terms of the real numbers, the rational numbers work just as well. In fact, any ordered, divisible, torsion free, Abelian group will do.

We present verified implementations of two quantifier elimination procedures: one due to Ferrante and Rackoff [6] and one due to Loos and Weispfenning [12].

5.1 Atoms

Type *atom* formalizes the normal forms explained above:

$$\textbf{datatype}\ atom = real < (real\ list)\ \mid\ real = (real\ list)$$

The second constructor argument is the list of coefficients [c_0,...,c_n] of the variables *0* to n — remember de Bruijn! Coefficient lists should be viewed as vectors and we define the usual vector operations on them:

$x *_s xs$ is the componentwise multiplication of a scalar x with a vector xs.
$xs + ys$ and $xs - ys$ are componentwise addition and subtraction of vectors.
$\langle xs,ys\rangle = (\sum (x,y) \leftarrow zip\ xs\ ys.\ x*y)$ is the inner product of two vectors, i.e. the sum over the componentwise products.

If the two vectors involved in an operation are of different length, the shorter one is padded with 0s (as in Obua's treatment of matrices [18]). We can prove all the algebraic properties we need, like $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$.

Now we instantiate locale *ATOM* just like for DLO in §4.1. The main function is the interpretation I_R of atoms, which is straightforward:

$$I_R\ (r < cs)\ xs = (r < \langle cs,xs\rangle) \qquad I_R\ (r = cs)\ xs = (r = \langle cs,xs\rangle)$$

5.2 Ferrante and Rackoff

Ferrante and Rackoff [6], inspired by Cooper [5], avoided DNF conversions by the test point method explained in §4. We have already explained the key idea of Ferrante and Rackoff in §4.2. If you replace $y{\downarrow}z$ in (1) by $(y+z)/2$ you almost obtain their algorithm. In principle any point between y and z works but $(y+z)/2$ also takes care of equalities: they lump E, L and U together (to be avoided in an implementation) but because $(y + y)/2 = y$ this recovers E. As their algorithm is well-known, we present its optimized and verified implementation right away:

$FR_1\ \varphi =$
(*let* $as = atoms_0\ \varphi$; $lbs = lbounds\ as$; $ubs = ubounds\ as$; $ebs = ebounds\ \varphi$;
  $intrs = [subst\ \varphi\ (between\ l\ u)\ .\ l \leftarrow lbs,\ u \leftarrow ubs]$;
in $list\text{-}disj\ (inf_-\ \varphi\ \cdot\ inf_+\ \varphi\ \cdot\ intrs\ @\ map\ (subst\ \varphi)\ ebs)$)

Except for the definition of *intrs* this looks identical to the definition of $interior_1$ in §4.3. However, all auxiliary functions are different: they operate on pairs (r, cs) which, under a valuation xs, represent the value $r + \langle cs,xs\rangle$. First the various bounds are extracted:

$lbounds\ as = [(r/c,\ (-1/c) *_s cs).\ (r < (c \cdot cs)) \leftarrow as,\ c{>}0]$
$ubounds\ as = [(r/c,\ (-1/c) *_s cs).\ (r < (c \cdot cs)) \leftarrow as,\ c{<}0]$
$ebounds\ as = [(r/c,\ (-1/c) *_s cs).\ (r = (c \cdot cs)) \leftarrow as,\ c{\neq}0]$

The intermediate point between two such points is easy:

$between\ (r,\ cs)\ (s,\ ds) = ((r + s)\ /\ 2,\ (1\ /\ 2) *_s (cs + ds))$

We need both ordinary substitution of $(r,\ cs)$ pairs

$asubst\ (r,\ cs)\ (s < d \cdot ds) = (s - d * r < d *_s cs + ds)$
$asubst\ (r,\ cs)\ (s = d \cdot ds) = (s - d * r = d *_s cs + ds)$
$asubst\ rcs\ a = a$

$subst\ \varphi\ rcs \equiv map_{fm}\ (asubst\ rcs)\ \varphi$

and substitution inf_- of $-\infty$ (and the analogous version inf_+ for ∞):

$inf_-\ (\varphi_1 \wedge \varphi_2) = and\ (inf_-\ \varphi_1)\ (inf_-\ \varphi_2)$
$inf_-\ (\varphi_1 \vee \varphi_2) = or\ (inf_-\ \varphi_1)\ (inf_-\ \varphi_2)$
$inf_-\ (A\ (r < c \cdot cs)) = (\mathsf{if}\ c < 0\ \mathsf{then}\ \top\ \mathsf{else\ if}\ 0 < c\ \mathsf{then}\ \bot\ \mathsf{else}\ A\ (r < cs))$
$inf_-\ (A\ (r = c \cdot cs)) = (\mathsf{if}\ c = 0\ \mathsf{then}\ A\ (r = cs)\ \mathsf{else}\ \bot)$

The remaining cases are the identity. This concludes the auxiliary functions.

5.3 Loos and Weispfenning

The method of infinitesimals described in §4.4 was inspired by the analogous method for linear real arithmetic proposed by Loos and Weispfenning [12] who also showed practical examples where it outperforms Ferrante and Rackoff. Yet this method seems relatively unknown in the literature. Its implementation eps_1 is textually identical to the one for DLO in §4.4. But the auxiliary functions differ. Luckily we have seen all of them already, except $subst_+$:

$asubst_+\ (r,\ cs)\ (s < d \cdot ds) =$
$(\mathsf{if}\ d = 0\ \mathsf{then}\ A\ (s < ds)$
$\ \mathsf{else\ let}\ u = s - d * r;\ v = d *_s cs + ds;\ lessa = A\ (u < v)$
$\quad \mathsf{in\ if}\ d < 0\ \mathsf{then}\ lessa\ \mathsf{else}\ lessa \vee A\ (u = v))$
$asubst_+\ rcs\ (r = d \cdot ds) = (\mathsf{if}\ d = 0\ \mathsf{then}\ A\ (r = ds)\ \mathsf{else}\ \bot)$
$asubst_+\ rcs\ a = A\ a$

$subst_+\ \varphi\ rcs \equiv amap_{fm}\ (asubst_+\ rcs)\ \varphi$

6 Presburger Arithmetic

Presburger arithmetic needs a divisibility (or congruence) predicate "|" to allow quantifier elimination. On the other hand we restrict our attention to $\leq$ because $i < j$ is equivalent with $i + 1 \leq j$. Thus all atoms are of the form $i \leq k_0 * x_0 + \cdots + k_n * x_n$ or $d \parallel i + k_0 * x_0 + \cdots k_n * x_n$, where $\parallel$ is $|$ or $\nmid$, and $d, i, k_0, \ldots, k_n \in \mathbb{Z}$ and $d > 0$. This becomes the **datatype**

$$atom = Le\ int\ (int\ list)\ |\ Dvd\ int\ int\ (int\ list)\ |\ NDvd\ int\ int\ (int\ list)$$

We have avoided infix constructors because they work less well for ternary operations. Atoms are interpreted w.r.t. a list of variables as usual:

$I_Z\ (Le\ i\ ks)\ xs = (i \leq \langle ks,xs \rangle)$
$I_Z\ (Dvd\ d\ i\ ks)\ xs = d \mid (i + \langle ks,xs \rangle)$
$I_Z\ (NDvd\ d\ i\ ks)\ xs = (\neg\ d \mid (i + \langle ks,xs \rangle))$

Note that we reuse the polymorphic vector, i.e. list operations like $\langle .,. \rangle$ introduced for linear real arithmetic: they are defined for arbitrary types with 0, + and $*$.

The parameters of locale *ATOM* are instantiated as follows. The interpretation of atoms is given by function I_Z above, their negation by

$neg_Z\ (Le\ i\ ks) = A\ (Le\ (1 - i)\ (-\ ks))$
$neg_Z\ (Dvd\ d\ i\ ks) = A\ (NDvd\ d\ i\ ks) \qquad neg_Z\ (NDvd\ d\ i\ ks) = A\ (Dvd\ d\ i\ ks)$

and their decrementation by

$$decr_Z\ (Le\ i\ ks) = Le\ i\ (tl\ ks)$$
$$decr_Z\ (Dvd\ d\ i\ ks) = Dvd\ d\ i\ (tl\ ks) \quad decr_Z\ (NDvd\ d\ i\ ks) = NDvd\ d\ i\ (tl\ ks)$$

Parameter $depends_0$ becomes $\lambda a.\ hd\text{-}coeff\ a \neq 0$ where

$$hd\text{-}coeff\ (Le\ i\ ks) = (\mathsf{case}\ ks\ \mathsf{of}\ [] \Rightarrow 0 \mid k \cdot x \Rightarrow k)$$
$$hd\text{-}coeff\ (Dvd\ d\ i\ ks) = (\mathsf{case}\ ks\ \mathsf{of}\ [] \Rightarrow 0 \mid k \cdot x \Rightarrow k)$$
$$hd\text{-}coeff\ (NDvd\ d\ i\ ks) = (\mathsf{case}\ ks\ \mathsf{of}\ [] \Rightarrow 0 \mid k \cdot x \Rightarrow k)$$

6.1 Cooper's Algorithm

Cooper's algorithm relies on Cooper's theorem [5] which holds provided all coefficients of x in $\phi(x)$ are 1 or -1 (or 0):

$$(\exists x.\ \phi(x)) \ = \ (\bigvee_{j \in (0,\delta-1)} \phi_{-\infty}(j) \vee \bigvee_{y \in L} \bigvee_{j \in (0,\delta-1)} \phi(y+j))$$

where δ is the lcm of all d such that $d \mid t$ or $d \nmid t$ occurs in $\phi(x)$ and t contains x, L is the set of lower bounds for x in $\phi(x)$, and $\phi_{-\infty}(j)$ is $\phi(x)$ where x has been replaced by $-\infty$ in all inequations and by j in all other atoms.

We start by setting all (non-zero) head coefficients to 1 or -1. This is achieved by multiplying each atom a (with non-zero head coefficient) with m/k, where m is the lcm of all (non-zero) head coefficients and k is a's head coefficient (assume $k > 0$ for simplicity). Now all (non-zero) head coefficients are m, we replace them by 1 and conjoin the atom $m \mid x_0$. This is what *hd-coeff1* does for an atom and *hd-coeff1* for a formula:

$$hd\text{-}coeff1\ m\ (Le\ i\ (k \cdot ks)) =$$
$$(\mathsf{if}\ k = 0\ \mathsf{then}\ Le\ i\ (k \cdot ks)$$
$$\ \mathsf{else\ let}\ m' = m\ div\ |k|\ \mathsf{in}\ Le\ (m' * i)\ (sgn\ k \cdot m' *_s ks))$$
$$hd\text{-}coeff1\ m\ (Dvd\ d\ i\ (k \cdot ks)) =$$
$$(\mathsf{if}\ k = 0\ \mathsf{then}\ Dvd\ d\ i\ (k \cdot ks)$$
$$\ \mathsf{else\ let}\ m' = m\ div\ k\ \mathsf{in}\ Dvd\ (m' * d)\ (m' * i)\ (1 \cdot m' *_s ks))$$
$$hd\text{-}coeff1\ m\ (NDvd\ d\ i\ (k \cdot ks)) =$$
$$(\mathsf{if}\ k = 0\ \mathsf{then}\ NDvd\ d\ i\ (k \cdot ks)$$
$$\ \mathsf{else\ let}\ m' = m\ div\ k\ \mathsf{in}\ NDvd\ (m' * d)\ (m' * i)\ (1 \cdot m' *_s ks))$$
$$hd\text{-}coeff1\ m\ a = a$$

$$hd\text{-}coeffs1\ \varphi =$$
$$(\mathsf{let}\ m = zlcms\ (map\ hd\text{-}coeff\ (atoms_0\ \varphi))$$
$$\ \mathsf{in}\ A\ (Dvd\ m\ 0\ [1]) \wedge map_{fm}\ (hd\text{-}coeff1\ m)\ \varphi)$$

The sign function *sgn* returns -1, 0, and 1 for negative, zero and positive arguments. Functions *zlcms* computes the positive lcm of a list of integers.

Now we start to implement Cooper's theorem. The substitution $\phi_{-\infty}(j)$ is implemented by the composition of

$inf_{\_}\ (\varphi_1 \wedge \varphi_2) = and\ (inf_{\_}\ \varphi_1)\ (inf_{\_}\ \varphi_2)$
$inf_{\_}\ (\varphi_1 \vee \varphi_2) = or\ (inf_{\_}\ \varphi_1)\ (inf_{\_}\ \varphi_2)$
$inf_{\_}\ (A\ (Le\ i\ (k \cdot ks))) =$
$(\mathsf{if}\ k < 0\ \mathsf{then}\ \top\ \mathsf{else\ if}\ 0 < k\ \mathsf{then}\ \bot\ \mathsf{else}\ A\ (Le\ i\ (0 \cdot ks)))$

$inf_{\_}\ \varphi = \varphi$

and ordinary substitution:

$asubst\ i'\ ks'\ (Le\ i\ (k \cdot ks)) = Le\ (i - k * i')\ (k *_s ks' + ks)$
$asubst\ i'\ ks'\ (Dvd\ d\ i\ (k \cdot ks)) = Dvd\ d\ (i + k * i')\ (k *_s ks' + ks)$
$asubst\ i'\ ks'\ (NDvd\ d\ i\ (k \cdot ks)) = NDvd\ d\ (i + k * i')\ (k *_s ks' + ks)$

$asubst\ i'\ ks'\ a = a$

$subst\ i\ ks\ \varphi \equiv map_{fm}\ (asubst\ i\ ks)\ \varphi$

The right-hand side of Cooper's theorem now becomes executable:

$cooper_1\ \varphi =$
$(\mathsf{let}\ as = atoms_0\ \varphi;\ d = zlcms(map\ divisor\ as);$
$\quad lbs = [(i,ks).\ Le\ i\ (k \cdot ks) \leftarrow as,\ k{>}0]$
$\mathsf{in}\ or\ (Disj\ [0..d - 1]\ (\lambda n.\ subst\ n\ []\ (inf_{\_}\ \varphi)))$
$\quad (Disj\ lbs\ (\lambda(i,ks).\ Disj\ [0..d - 1]\ (\lambda n.\ subst\ (i + n)\ (-ks)\ \varphi))))$

where $divisor\ (Dvd\ d\ \_\ \_) = divisor(NDvd\ d\ \_\ \_) = d$, $divisor\ (Le\ \_\ \_) = 1$ and $Disj\ us\ f \equiv list\text{-}disj\ (map\ f\ us)$. The lower bounds *lbs* are computed directly rather than by an auxiliary function.

The two phases of Cooper's algorithm are simply composed and lifted:

$cooper = lift\text{-}nnf\text{-}qe\ (cooper_1 \circ hd\text{-}coeffs1)$

6.2 Correctness

There is a slight complication we have glossed over so far. We want to exclude the atoms *Dvd 0 i ks* and *NDvd 0 i ks* because they behave anomalously and the algorithm does not generate them either. Catering for them would complicate the algorithm with case distinctions. In order to restrict attention to a subset of *normal* atoms, locale *ATOM* in fact has another parameter not mentioned so far: $anormal :: \alpha \Rightarrow bool$ with the axioms

$$anormal\ a \Longrightarrow \forall\, b{\in}atoms\ (aneg\ a).\ anormal\ b$$
$$\neg\ depends_0\ a \Longrightarrow anormal\ a \Longrightarrow anormal\ (decr\ a)$$

In words: negation and decrementation do not lead outside the normal atoms. These axioms allow to show the following refined version of Lemma 2 (inside *ATOM*), where $normal\ \varphi = (\forall\, a{\in}atoms\ \varphi.\ anormal\ a)$:

Lemma 7. *If* $qe \in |nqfree| \to |qfree|$ *and* $qe \in |nqfree| \cap |normal| \to |normal|$ *and for all* φ *and* xs*:* $normal\ \varphi \wedge nqfree\ \varphi \Longrightarrow I\ (qe\ \varphi)\ xs = (\exists\, x.\ I\ \varphi\ (x \cdot xs))$*, then* $normal\ \varphi$ *implies* $I\ (lift\text{-}nnf\text{-}qe\ qe\ \varphi)\ xs = I\ \varphi\ xs$*.*

In the instantiation of *ATOM* for Presburger arithmetic parameter *anormal* becomes $\lambda a.\ \mathit{divisor}\ a \neq 0$. The above lemma is instantiated with $\mathit{cooper}_1 \circ \mathit{hd\text{-}coeffs1}$ for *qe* and its premises are discharged by the detailed but familiar correctness arguments for Cooper's algorithm. We obtain the corollary $\mathit{normal}\ \varphi \Longrightarrow I\ (\mathit{cooper}\ \varphi)\ \mathit{xs} = I\ \varphi\ \mathit{xs}$. Of course $\mathit{qfree}\ (\mathit{cooper}\ \varphi)$ is also proved.

7 Related work

The literature on decision procedures for linear arithmetic is vast. We concentrate on formally verified algorithms.

Nipkow [15] presents the generic framework of §3 in detail but concentrates on non-elementary DNF-based procedures. Chaieb and Nipkow [4] present a reflective implementations of Cooper's algorithm. But they lack the generic framework and they use special purpose data structures for terms instead of relying on lists as we do. As a result some of their functions are considerably more complicated than ours and theorems and proofs are littered with linearity assumptions that are implicit in our list representation. Hence they can only present part of their implementation. Chaieb [3] presents a verified combination of Ferrante-Rackoff and Cooper. Norrish [17] was the first to implement a proof-producing version of Cooper's algorithm in a theorem prover. Similar implementation of QE for complex numbers and for real closed fields are reported by Harrison [10] and McLaughlin [14]. The CAD QE procedure for real closed fields has been reflected but only partly verified by Mahboubi [13] in Coq.

Acknowledgment Amine Chaieb alerted me to the infinitesimal approach [12]. Discussions with him and Jeremy Avigad were very helpful.

References

1. C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. Borwein and W. Farmer, editors, *Mathematical Knowledge Management*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.
2. R. S. Boyer and J. S. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.
3. A. Chaieb. Verifying mixed real-integer quantifier elimination. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 528–540. Springer, 2006.
4. 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 (LPAR 2005)*, volume 3835 of *LNCS*, pages 367–380. Springer, 2005.
5. D. 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.

6. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Computing*, 4:69–76, 1975.
7. G. Gonthier. A computer-checked proof of the four-colour theorem. http://research.microsoft.com/~gonthier/4colproof.pdf.
8. F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *LNCS*, pages 160–174. Springer, 2007.
9. J. Harrison. *Introduction to Logic and Automated Theorem Proving*. Cambridge University Press. Forthcoming.
10. J. Harrison. Complex quantifier elimination in HOL. In R. Boulton and P. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, pages 159–174. Division of Informatics, University of Edinburgh, 2001.
11. C. Langford. Some theorems on deducibility. *Annals of Mathematics (2nd Series)*, 28:16–40, 1927.
12. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36:450–462, 1993.
13. A. Mahboubi. *Contributions à la certification des calculs sur* $\mathbb{R}$ *: théorie, preuves, programmation*. PhD thesis, Université de Nice, 2006.
14. S. McLaughlin and J. Harrison. A proof-producing decision procedure for real arithmetic. In R. Nieuwenhuis, editor, *Automated Deduction — CADE-20*, volume 3632 of *LNCS*, pages 295–314. Springer, 2005.
15. T. Nipkow. Reflecting quantifier elimination for linear arithmetic. In O. Grumberg, T. Nipkow, and C. Pfaller, editors, *Formal Logical Methods for System Security and Correctness*, pages 245–266. IOS Press, 2008.
16. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
17. M. Norrish. Complete integer decision procedures as derived rules in HOL. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2003*, volume 2758 of *LNCS*, pages 71–86. Springer, 2003.
18. S. Obua. Proving bounds for real linear programs in Isabelle/HOL. In J. Hurd, editor, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 227–244. Springer, 2005.
19. V. Weispfenning. The complexity of linear problems in fields. *J. Symbolic Computation*, 5:3–27, 1988.