Verifying and reflecting quantifier elimination for Presburger arithmetic

Amine Chaieb and Tobias Nipkow

Institut für Informatik Technische Universität München

Abstract. We present an implementation and verification in higherorder logic of Cooper's quantifier elimination for Presburger arithmetic. Reflection, i.e. the direct execution in ML, yields a speed-up of a factor of 200 over an LCF-style implementation and performs as well as a decision procedure hand-coded in ML.

1 Introduction

This paper presents a formally verified quantifier elimination procedure for Presburger arithmetic (\mathcal{PA}) in higher-order logic. There are three approaches to decision procedures in theorem provers: unverified code (which we ignore), *LCF-style* proof procedures programmed in a metalanguage (ML) that invoke the inference rules of the kernel, and *reflection*, where the decision procedure is formalized and proved correct inside the system and is executed not by inference but by direct computation.

The LCF-style requires no formalization of the meta-theory but has a number of disadvantages: (a) it requires intimate knowledge of the internals of the underlying theorem prover (which makes it very unportable); (b) there is no way to check at compile type if the proofs will really compose (which easily leads to run time failure and thus incompleteness); (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 for \mathcal{PA} may require super exponential space [7, 16]).

For all these reasons we have formalized and verified Cooper's quantifier elimination procedure for \mathcal{PA} [5]. Our development environment is Isabelle/HOL [14]. An experimental feature allows reflective extensions of the kernel: computations of ML code generated from HOL functions [3] are accepted as equality proofs. Such extensions are sound provided the code generator is correct. Coq uses a fast internal λ -calculus evaluator for the same purpose [8]. We found that reflection leads to a substantial performance improvement. This is especially marked when proof objects [2] are involved: reflective subproofs are of constant size, which is particularly important for proof carrying code applications, where the size of full \mathcal{PA} proofs is prohibitive.

The main contributions of our work are: (a) the first-time formalization and verification of Cooper's decision procedure in a theorem prover; (b) the most substantial (5000 lines) application of reflection in any theorem prover to date (as far as we are aware); (c) a formalization that is easily portable to other theorem provers supporting reflection (in contrast to LCF-tactics); (d) performance figures that show a speed-up of up to 200 w.r.t. a comparable LCF-style implementation; (e) a first demonstration of reflection in Isabelle/HOL. We also provide a nice example of how reflection allows to formalize duality/symmetry arguments based on syntax (function mirror in 4.2).

Related work \mathcal{PA} has first been proven decidable by Presburger [17] whose (inefficient) algorithm was improved by Cooper [5]. Harrison [12] implemented Cooper's procedure as an oracle as well as partially reflected in HOL Light. In [4] we presented an LCF-style implementation of Cooper's algorithm for PA, which is our point of reference. Harrison [10] has also studied the general issue of reflection in LCF-like theorem provers and bemoans the lack of a natural example where reflection yields a speed-up of more than a constant factor. This is true for \mathcal{PA} as well, but a constant factor of 200 over an LCF-style tactic is worth it. Norrish [15] discusses implementations for both Cooper's algorithm (in tatic style) and Omega [18] (checking a reflected "proof trace"). Pierre Crgut [6] presents a reflective version of the Omega test written for Coq, where an optimized proof trace is interpreted to solve the goal. Unlike the other references his implementation only deals with quantifier-free PA and is incomplete. Presburger's original algorithm has been formalized in Coq by Laurent Thry and is available on the Coq web site.

The problem of programming errors in decision procedures has recently been addressed by several authors using dependent types [13, 1]. But it seems unlikely that anything as complex as \mathcal{PA} can be dealt with automatically in such a framework. Nor does this approach guarantee completeness: missing cases and local proofs that fail are not detected.

Notation 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. The data type α option with the constructors $\perp : \alpha$ option and $\lfloor . \rfloor : \alpha \rightarrow \alpha$ option models computations that can fail.

The rest of this paper is structured as follows. In 2 we give a brief overview of reflection. The actual decision procedure and its verification is presented in 3 and 4. In 5 we discuss some design decisions and alternatives. Performance results are shown in 6.

2 Reflection

2.1 An informal introduction

Reflection means to perform a proof step by computation inside the logic. However, inside the logic it is not possible to write functions by pattern matching over the syntax of terms or formulae 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 also called the *shadow syntax* [11]. Let us call this type *rep*, the representation.

Then there are two functions: *interp*, a function in the logic, maps an element of *rep* to the formula it represents; *convert*, an ML function, maps a formula to its representation. The two functions should be inverses of each other: taking the ML representation of a formula P and applying *convert* to it yields an ML representation of a term p of type *rep* such that the theorem *interp* p = P can be proved by by rewriting with the equations for *interp*.

Typically, the formalized proof step is some equivalence P = P' where P is given and P' is some simplified version of P (e.g. the elimination of a quantifier). This transformation is now expressed as a recursive function simp of type $rep \rightarrow rep$ and it is proved (typically by induction on rep) that simp preserves the interpretation:

$interp \ p = interp(simp \ p).$

To apply this theorem to a given formula P we compute (in ML) p = convert P, substitute it into our theorem, and compute the value P' of interp(simp p). The latter step should be done as efficiently as possibly. In our case it is performed by an ML computation using the code automatically generated from the defining equations for simp and interp. This yields the theorem interp(simp p) = P'. Combining it (by symmetry and transitivity) with $interp \ p = P$ and $interp \ p = interp(simp \ p)$ we obtain the theorem P = P'.

2.2 Reflection of PA

 \mathcal{PA} is reflected as follows. The syntax is represented by the data types ι for integer expressions and ϕ for formulae.

$$\begin{array}{l} \text{datatype } \iota = \widehat{int} \mid v_{nat} \mid - \iota \mid \iota + \iota \mid \iota - \iota \mid \iota * \iota \\ \text{datatype } \phi = \iota < \iota \mid \iota > \iota \mid \iota \leq \iota \mid \iota \geq \iota \mid \iota \geq \iota \mid \iota = \iota \mid \iota \ \mathbf{dvd} \ \iota \\ \quad \mid \mathbf{T} \mid \mathbf{F} \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \rightarrow \phi \mid \phi = \phi \mid \exists \phi \mid \forall \phi \end{array}$$

The bold symbols $+, \leq, \wedge$ etc are constructors and reflect their counterparts $+, \leq, \wedge$ etc in the logic. The integer constant *i* in the logic is represented by the term \hat{i} . Bound variables are represented by de Bruijn indices: v_n represents the bound variable with index *n* (a natural number). Hence quantifiers need not carry variable names.

Throughout the paper p and q are of type ϕ .

$$\begin{split} & \begin{bmatrix} \widehat{l} \end{bmatrix}_{\iota^{is}}^{is} &= True \\ & \llbracket v_n \rrbracket_{\iota}^{is} &= is!n \\ & \llbracket a + b \rrbracket_{\iota}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} - \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} &= \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} = \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} = \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} = \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} = \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is} \\ & \llbracket a + b \rrbracket_{\iota^{is}}^{is} + \llbracket a \rrbracket_{\iota^{is}}^{is} + \llbracket b \rrbracket_{\iota^{is}}^{is$$

Fig. 1. Semantics of the shadow syntax

The interpretation functions $(\llbracket . \rrbracket_i \text{ and } \llbracket . \rrbracket)$ in Fig. 1 map the representations back into logic. They are parameterized by an environment *is* which is a list of integer expressions. The de Bruijn index v_n picks out the n^{th} element from that list.

The definition of ι -terms is too liberal since it allows to express nonlinear terms. Hence we will impose conditions during verification which guarantee that terms have certain syntactic shapes.

3 Quantifier elimination

A generic quantifier elimination function is implemented by $\operatorname{\mathsf{qelim}}_{\phi}$ (Fig. 2). Its parameter qe is supposed to eliminate a single \exists and $\operatorname{\mathsf{qelim}}_{\phi}$ applies qe to all quantified subformulae in a bottom-up fashion. We allow quantifier elimination to fail, i.e. return \bot . This is necessary in case the input

$\operatorname{qelim}_{\phi} qe$ ((p) = (p)	$\neg_{\perp}(qe^{\perp}(\neg_{\perp}(qelim_{\phi} \ qe \ p)))$
$\operatorname{qelim}_{\phi} qe$ (Ξ	$\exists p) =$	$qe^{\perp}(qelim_{\phi} \ qe \ p)$
$\operatorname{qelim}_{\phi} qe (p$	$(\wedge q) =$	$(\operatorname{qelim}_{\phi} qe p) \wedge_{\perp} (\operatorname{qelim}_{\phi} qe p)$
$\operatorname{qelim}_{\phi} qe (p$	$(\lor q) =$	$(\operatorname{qelim}_\phi \ qe \ p) \lor_\perp (\operatorname{qelim}_\phi \ qe \ p)$
$\operatorname{qelim}_{\phi} qe (p$	$p \rightarrow q) =$	$(\operatorname{qelim}_\phi qe p) \rightarrow_\perp (\operatorname{qelim}_\phi qe p)$
$\operatorname{qelim}_{\phi} qe (p$	p = q) =	$(\operatorname{qelim}_{\phi} qe p) =_{\perp} (\operatorname{qelim}_{\phi} qe p)$
$\operatorname{qelim}_\phi qe p$	=	$\lfloor p \rfloor$

Fig. 2. Quantifier elimination for ϕ -formulae

formula is not linear, i.e. involves multiplication by more than just a constant. To deal with failure we define two combinators for lifting arbitrary nary functions f to f^{\perp} and f_{\perp} :

$$f^{\perp} [x_1] \dots [x_n] = f x_1 \dots x_n$$
$$f_{\perp} [x_1] \dots [x_n] = [f x_1 \dots x_n]$$

If any of the arguments are \perp , f^{\perp} and f_{\perp} return \perp .

Let qfree p (not shown) formalize that p is quantifier-free. We can prove by structural induction that if qe takes a quantifier-free formula qand returns a quantifier-free formula q' equivalent to $\exists q$, then $\operatorname{\mathsf{qelim}}_{\phi} qe$ is a quantifier-elimination procedure:

$$(\forall q, q', is. \text{ qfree } q \land qe \ q = \lfloor q' \rfloor \to \text{qfree } q' \land \llbracket \exists \ q \rrbracket^{is} = \llbracket q' \rrbracket^{is})$$

$$\to \text{qelim}_{\phi} \ qe \ p = \lfloor p' \rfloor \to \text{qfree } p' \land \llbracket p \rrbracket^{is} = \llbracket p' \rrbracket^{is}.$$
(1)

Note that qe must eliminate the innermost bound variable v_0 , otherwise $[\exists q]^{is} = [q']^{is}$ will not hold.

The goal of 4 is to present cooper, an instance of qe fulfilling the premise of (1).

4 Cooper's algorithm

Like many decision procedures, Cooper's algorithm [5] for eliminating one \exists follows a simple scheme:

- Normalization of input formula (4.1).
- Calculation of some characteristic data from the formula (4.2).
- Correctness theorem proving that $\exists p$ is semantically equivalent to a simpler formula p' involving the data from the previous step (Cooper's theorem in 4.3).
- Construction of p' (4.4).

4.1 Normalization

Normalization goes trough three steps: the N-step puts the formula into NNF (negation normal form), the L-step linearizes the formula and the U-step sets the coefficients of v_0 to $\widehat{1}$ or $\widehat{-1}$.

The N-step We omit the straightforward implementation of $\operatorname{nnf} : \phi \to \phi$ and $\operatorname{isnnf} : \phi \to \operatorname{bool}$. Property $\operatorname{isnnf} p$ expresses that p is in NNF and that all atoms are among \leq , = and dvd and that negations only occur in front of dvd or =. We prove that nnf is correct and that it implies quantifier-freedom:

$$\llbracket p \rrbracket^{is} = \llbracket \mathsf{nnf} \ p \rrbracket^{is} \qquad \mathsf{isnnf}(\mathsf{nnf} \ p) \qquad \mathsf{isnnf} \ p \to \mathsf{qfree} \ p$$

The L-step An ι -term t is *linear* if it has the form

$$\widehat{c_1} * v_{i_1} + \dots + \widehat{c_n} * v_{i_n} + \widehat{c_{n+1}}$$

where $n \in \mathbb{N}$, $i_1 < \cdots < i_n$ and $\forall j \leq n.c_j \neq 0$. Note that $\widehat{c_{n+1}}$ is always present even if $c_{n+1} = 0$. The implementation is easy:

$$\begin{array}{lll} \operatorname{islinn}_{\iota} n_0 \widehat{i} & = True \\ \operatorname{islinn}_{\iota} n_0 (\widehat{i} * v_n + r) & = i \neq 0 \wedge n_0 \leq n \wedge \operatorname{islinn}_{\iota} (n+1) r \\ \operatorname{islinn}_{\iota} n_0 t & = False \\ \operatorname{islin}_{\iota} t & = \operatorname{islinn}_{\iota} 0 t \end{array}$$

A formula p is linear (islin $_{\phi}$ p) if it is in NNF, all ι -terms occurring in it are linear, and its atoms are of the form $t \leq \hat{0}, t = \hat{0}$ or $\hat{d} \, \mathbf{dvd} \, t$ where $d \neq 0$. The formal definition is omitted.

The goal of the L-step is to transform a formula into an equivalent linear one. Due to the unrestricted use of * in the input syntax ι this may fail. Function \lim_{ι} (Fig. 3) tries to linearize an ι -term using \lim_{+} , \lim_{+} and \lim_{-} . These operate on linear ι -terms, preserve linearity and behave semantically like addition, multiplication by a constant integer and multiplication by -1, respectively. This is expressed by the following theorems provable by induction:

$$\begin{aligned} \operatorname{islin}_{\iota} a \wedge \operatorname{islin}_{\iota} b &\to \operatorname{islin}_{\iota}(\operatorname{lin}_{+} a b) \wedge (\llbracket \operatorname{lin}_{+} a b \rrbracket_{\iota}^{is} = \llbracket a + b \rrbracket_{\iota}^{is}) \\ \operatorname{islin}_{\iota} a &\to \operatorname{islin}_{\iota}(\operatorname{lin}_{*} i a) \wedge (\llbracket \operatorname{lin}_{*} i a \rrbracket_{\iota}^{is} = \llbracket \widehat{i} * a \rrbracket_{\iota}^{is}) \\ \operatorname{islin}_{\iota} a &\to \operatorname{islin}_{\iota}(\operatorname{lin}_{-} a) \wedge (\llbracket \operatorname{lin}_{-} a \rrbracket_{\iota}^{is} = \llbracket - a \rrbracket_{\iota}^{is}) \end{aligned}$$

The implementations of lin_{*} and lin₋ are omitted for space limitations.

```
\lim_{l \to \infty} (\widehat{k} * v_n + r) (\widehat{l} * v_m + s) =

if n = m then
   if k+l=0 then \lim_{l \to \infty} r s else \widehat{k+l} * v_n + \lim_{l \to \infty} r s
else if n \leq m then \hat{k} * v_n + \lim_{l \to \infty} r (\hat{l} * v_m + s)
   else \hat{l} * v_m + \lim_{l \to \infty} (\hat{k} * v_n + r) s

\lim_{k \to \infty} (\widehat{k} * v_n + r) \ \widehat{b} = \widehat{k} * v_n + \lim_{k \to \infty} r \ \widehat{b}

lin_{+} \widehat{a} (\widehat{l} * v_{n} + s) = \widehat{l} * v_{n} + lin_{+} s \widehat{a}

\lim_{+} \widehat{k} \ \widehat{l} = \widehat{k+l}
\lim_{\iota} \widehat{c} = \lfloor \widehat{c} \rfloor
\lim_{\iota} v_n = \lfloor (\widehat{1} * v_n + \widehat{0}) \rfloor
\lim_{\iota} (-a) = \lim_{-\perp} (\lim_{\iota} a)
\lim_{\iota} (a+b) = \lim_{+\perp} (\lim_{\iota} a) (\lim_{\iota} b)
\lim_{\iota} (a - b) = \lim_{+ \perp} (\lim_{\iota} a) (\lim_{\iota} (-b))
\mathsf{lin}_{\iota} \ (a \ast b) =
case (\lim_{\iota} a, \lim_{\iota} b) of
      (\lfloor \widehat{c} \rfloor, \lfloor b' \rfloor) \Rightarrow \lfloor \mathsf{lin}_* \ c \ b' \rfloor
      (\lfloor a' \rfloor, \lfloor \widehat{c} \rfloor) \Rightarrow \lfloor \lim_{*} c \ a' \rfloor
      (x,y) \Rightarrow \bot
```

Fig. 3. linearization of ι -terms

Linearization of ϕ -formulae (lin $_{\phi}$, not shown) lifts lin $_{\iota}$. We have proved that it also preserves semantics and linearizes its input:

isnnf
$$p \wedge \lim_{\phi} p = \lfloor p' \rfloor \rightarrow \llbracket p \rrbracket^{is} = \llbracket p' \rrbracket^{is} \wedge \operatorname{islin}_{\phi} p'$$

Since full linearization is not really part of Presburger arithmetic, we keep matters simple and do not try to cancel arbitrary monomials: $\lim_{\iota} (v_0 * v_0 - v_0 * v_0) = \bot$ although one could also return $\lfloor \hat{0} \rfloor$. Such simplifications could be performed by a specialized algebraic preprocessor.

The U-step The key idea in this step is to multiply the terms occurring in atoms by appropriate constants such that the (absolute values of) coefficients of v_0 are the same everywhere, e.g. the *lcm* of all coefficients of v_0 . The equivalence

$$(\exists x. \ P(l \cdot x)) = (\exists x. \ l \ \mathsf{dvd} \ x \land P(x)). \tag{2}$$

will allow us to obtain a formula where all coefficients of v_0 are $\hat{1}$ or $-\hat{1}$. Function lcm_{ϕ} takes a formula p and computes $lcm\{c \mid \hat{c} * v_0 \text{ occurs in } p\}$. Predicate alldvd l p checks if all coefficients of v_0 in p divide l. Both functions are defined in the following table where lcm computes the positive least common multiple of two integers.

p	$lcm_{\phi} p$	alldvd $l p$	
$\widehat{c} * v_0 + r \le \widehat{z}$	c	$c \; dvd \; l$	
$\widehat{c} \ast v_0 + r = \widehat{z}$	c	$c \; dvd \; l$	
$\widehat{d} \operatorname{dvd} \widehat{c} * v_0 + r$	c	$c \; dvd \; l$	
$\neg p$	$lcm_{\phi} p$	alldvd $l p$	
$p \wedge q$	$\operatorname{lcm} (\operatorname{lcm}_{\phi} p) (\operatorname{lcm}_{\phi} q)$	$(alldvd\ l\ p) \land (alldvd\ l\ q)$	
$p \lor q$	$lcm\ (lcm_{\phi}\ p)\ (lcm_{\phi}\ q)$	$(alldvd\ l\ p) \land (alldvd\ l\ q)$	
-	1	True	

The correctness of these functions is expressed by the following theorem:

 $\mathsf{islin}_{\phi} p \rightarrow \mathsf{alldvd} (\mathsf{lcm}_{\phi} p) p \wedge \mathsf{lcm}_{\phi} p > 0$

The main part of the U-step is done by the function adjust. It takes a positive integer l and a linear formula p (assuming that alldvd l p holds) and produces a linear formula p' s.t. the coefficients of v_0 are set to either $\hat{1}$ or $-\hat{1}$. Function unity performs the U-step:

unity p =let $l = \text{lcm}_{\phi} p$; p' = adjust l p in if l = 1 then p' else $(\hat{l} \text{ dvd } \hat{1} * v_0 + \hat{0}) \land p'$

The resulting formula is said to be unified (unified p'). We omit the definition of adjust and unified. Note that unified $p \to \operatorname{islin}_{\phi} p$. We can prove that adjust preserves semantics and its result is unified

$$\begin{split} & \text{islin}_{\phi} \ p \land \text{alldvd} \ l \ p \land l > 0 \rightarrow \\ & \llbracket p \rrbracket^{i \cdot i s} = \llbracket \text{adjust} \ l \ p \rrbracket^{(l \cdot i) \cdot i s} \land \text{unified}(\text{adjust} \ l \ p) \end{split}$$

and with (2) the correctness of unity follows:

$$\mathsf{islin}_{\phi} \ p \ \to \ [\![\exists \ p]\!]^{is} = [\![\exists \ (\mathsf{unity} \ p)]\!]^{is} \land \mathsf{unified}(\mathsf{unity} \ p) \tag{3}$$

4.2 Calculation

In the next subsection we need to compute for a given p a pair of a set (represented as a list) of coefficients in p and a modified version of p. More precisely, we need to compute (bset p, p_{-}) or (aset p, p_{+}), which are dual to each other. Fig. 4 shows how to perform these computations recursively and it should be seen as the definition of four functions bset, aset, minusinf and plusinf. We use p_{-} and p_{+} as shorthands for minusinf pand plusinf p. Before we start proving properties about bset and minusinf we formalize the duality between (bset p, p_{-}) and (aset p, p_{+}). Theorems about bset and minusinf will then yield theorems about aset and plusinf. Syntactically the duality is expressed by the function mirror (Fig. 5) which negates all coefficients of v_0 . The following intuitive relationships between a formula and its mirrored version can be proved:

unified
$$p \rightarrow [\![p]\!]^{i \cdot i s} = [\![mirror \ p]\!]^{(-i) \cdot i s} \wedge \text{unified (mirror } p)$$

 $[\![\exists \ p]\!]^{i s} = [\![\exists \ (mirror \ p)]\!]^{i s}$ (4)

p	aset p	bset p	p_{-}	p_+
$q \wedge r$	aset $q @$ aset r	bset $q @$ bset r	$q \wedge r$	$q_+ \wedge r_+$
$q \lor r$	aset $q @$ aset r	$bset\; q @ bset\; r$	$q \lor r$	$q_+ \lor r_+$
$\widehat{1} \ast v_0 + a \le \widehat{0}$	$[-a, -a+\widehat{1}]$	$[-a-\widehat{1}]$	F	T
$\widehat{-1} \ast v_0 + a \le \widehat{0}$	$[a-\widehat{1}]$	$[a, a - \widehat{1}]$	T	F
$\widehat{1} \ast v_0 + a = \widehat{0}$	$[-a+\hat{1}]$	$[-a-\widehat{1}]$	F	\boldsymbol{F}
$\widehat{-1} \ast v_0 + a = \widehat{0}$	$[a+\widehat{1}]$	$[a-\widehat{1}]$	${oldsymbol{F}}$	F
$\neg \hat{1} \ast v_0 + a = \hat{0}$	[-a]	[-a]	T	T
$\neg \widehat{-1} * v_0 + a = \widehat{0}$	[a]	[a]	T	T
-	[]	[]	p	p

Fig. 4. Definition of aset p, bset p, p_- and p_+

```
\begin{array}{ll} \mbox{mirror} (\widehat{c} \ast v_0 + r \leq \widehat{z}) &= (\widehat{-c} \ast v_0 + r \leq \widehat{z}) \\ \mbox{mirror} (\widehat{c} \ast v_0 + r = \widehat{z}) &= (\widehat{-c} \ast v_0 + r = \widehat{z}) \\ \mbox{mirror} (\widehat{d} \mbox{ dvd } \widehat{c} \ast v_0 + r) &= (\widehat{d} \mbox{ dvd } \widehat{-c} \ast v_0 + r) \\ \mbox{mirror} (\neg \widehat{d} \mbox{ dvd } \widehat{c} \ast v_0 + r) &= (\neg \widehat{d} \mbox{ dvd } \widehat{-c} \ast v_0 + r) \\ \mbox{mirror} (\neg \widehat{c} \ast v_0 + r = \widehat{z}) &= (\neg \widehat{-c} \ast v_0 + r \leq \widehat{z}) \\ \mbox{mirror} (p \land q) &= (\mbox{mirror} \ l \ p) \land (\mbox{mirror} \ l \ q) \\ \mbox{mirror} p &= p \end{array}
```

Fig. 5. Mirroring a formula

Furthermore we have the following dualities:

$$islin_{\phi} \ p \to \llbracket plusinf \ p \rrbracket^{i \cdot is} = \llbracket minusinf(mirror \ p) \rrbracket^{(-i) \cdot is}$$

unified $p \to aset \ p = map \ lin_{bset} \ (mirror \ p))$ (5)

We will also need to compute $\delta_p = lcm\{d \mid \hat{d} \text{ dvd } \hat{c} * v_0 + r \text{ occurs in } p\}$. Its definition is very similar to that of $lcm_{\phi} p$. Finally let the predicate alldvd_{dvd} l p be the analogue of alldvd l p which ensures $islin_{\phi} p \rightarrow$ alldvd_{dvd} $\delta_p p$. The definition of both functions is obvious and omitted.

4.3 Cooper's theorem

Our proof sketch of Cooper's theorem (10) follows [15]. The conclusion of Cooper's theorem is of the form $A = (B \lor C)$ and we prove $B \to A$, $C \to A$ and $A \land \neg B \to C$. We first prove (by induction on p) that any unified p behaves exactly like minusinf p for values that are small enough, cf. (6), and that this behaviour is periodic, cf. (7).

unified
$$p \to \exists z. \forall x. x < z \to (\llbracket p \rrbracket^{x \cdot is} = \llbracket \text{minusinf } p \rrbracket^{x \cdot is})$$
 (6)

unified
$$p \to \forall x, k. [[minusinf p]]^{x \cdot is} = [[minusinf p]]^{(x-k \cdot \delta_p) \cdot is}$$
 (7)

Using (6) and (7) we can prove the first implication (8), i.e. any witness j for p_{-} provides a witness for p. According to (7) we can keep on decreasing j by δ until we reach the limit z of (6). This proof is based on induction over integers bounded from above. Note also that (8) holds for all d.

unified
$$p \land (\exists j \in \{1..d\}. \llbracket \text{minusinf } p \rrbracket^{j \cdot j s}) \to \llbracket \exists p \rrbracket^{j s}$$
 (8)

The second implication is trivial: given $b \in \{\{\text{bset } p\}\}$ and $j \in \{1..\delta_p\}$ such that $[\![p]\!]^{[\![i\cdot is]\!]_{\iota}^b+j}$ we have a witness for p. If there is no such b and j then p behaves periodically and hence any witness for p must be a witness for p_- . Hence (9) proves with (6) and (7) the last implication and Cooper's theorem (10) follows directly using (8).

unified
$$p \to \forall x. \neg (\exists j \in \{\!\!\{1..\delta_p\}\!\!\}. \exists b \in \{\!\!\{\mathsf{bset } p\}\!\!\}. [\![p]\!]^{([\![b]\!]_{\iota}^{i\cdot is}+j)\cdot is}) \to [\![p]\!]^{x\cdot is} \to [\![p]\!]^{(x-\delta_p)\cdot is}$$
(9)

unified
$$p \to (\llbracket \exists p \rrbracket^{is} = ((\exists j \in \{1..\delta_p\}.\llbracket minusinf p \rrbracket^{j\cdot is}) \lor (\exists j \in \{1..\delta_p\}.\exists b \in \{ \text{bset } p \}.\llbracket p \rrbracket^{(\llbracket b \rrbracket^{i\cdot is}_{\iota} + j)\cdot is})))$$
 (10)

This expresses that an existential quantifier is equivalent with a finite disjunction. The latter is still expressed with existential quantifiers, but we will now replace them by executable functions.

4.4 The decision procedure

In order to compute the rhs of Cooper's theorem (10) we need substitution for v_0 in ι -terms (subst_{ι}) and ϕ -formulae (subst_{ϕ}) such that

$$\llbracket \text{subst}_{\iota} \ r \ t \rrbracket_{\iota}^{i \cdot is} = \llbracket t \rrbracket_{\iota}^{\llbracket r \rrbracket_{\iota}^{j \cdot is} \cdot is}$$
$$\llbracket \text{subst}_{\phi} \ r \ p \rrbracket^{i \cdot is} = \llbracket p \rrbracket^{\llbracket r \rrbracket_{\iota}^{i \cdot is} \cdot is}$$

Let $\mathsf{nov0}_{\iota} t$ and $\mathsf{nov0}_{\phi} p$ express that v_0 does not occur in t and p, and let $\mathsf{decr}_{\iota} t$ and $\mathsf{decr}_{\phi} p$ denote t and p where all variable indices are decremented by one. The implementation of subst_{ι} , subst_{ϕ} , $\mathsf{nov0}_{\iota}$, $\mathsf{nov0}_{\phi}$, decr_{ι} and decr_{ϕ} is simple and omitted. The following properties are easy:

$$\begin{aligned} \mathsf{nov0}_{\iota} \ t &\to \mathsf{nov0}_{\iota} \ (\mathsf{subst}_{\iota} \ t \ r) \land \mathsf{nov0}_{\phi} \ (\mathsf{subst}_{\phi} \ t \ p) \\ \mathsf{nov0}_{\iota} \ t &\to \llbracket t \rrbracket_{\iota}^{i \cdot i s} = \llbracket \mathsf{decr}_{\iota} \ t \rrbracket_{\iota}^{i s} \\ \mathsf{nov0}_{\phi} \ p &\to \llbracket p \rrbracket^{i \cdot i s} = \llbracket \mathsf{decr}_{\phi} \ p \rrbracket^{i s} \end{aligned}$$

$$\begin{split} & \mathsf{explode}_{\vee} ~~[]~p = F \\ & \mathsf{explode}_{\vee}~(i \cdot is)~p = \\ & \mathsf{case}~(\mathsf{simp}~(\mathsf{subst}_{\phi}~i~p),\mathsf{explode}_{\vee}~is~p)~\mathsf{of} \\ & (T,_) \Rightarrow T \\ & (F,p_{is}) \Rightarrow p_{is} \\ & (_,T) \Rightarrow T \\ & (p_i,F) \Rightarrow p_i \\ & (p_i,p_{is}) \Rightarrow p_i \lor p_{is} \end{split}$$

Fig. 6. Generate disjunctions

To generate the disjunction $\bigvee_{t \in \{\!\!\{ts\}\!\!\}} \operatorname{subst}_{\phi} t p$ we use $\operatorname{explode}_{\bigvee} ts p$ (Fig. 6). Function simp evaluates ground atoms and performs simple propositionsal simplifications. We prove

 $\begin{array}{l} \mathsf{qfree} \ p \land (\forall t \in \{\!\!\{ts\}\!\!\}.\mathsf{nov0}_\iota \ t) \rightarrow \\ \mathsf{nov0}_\phi(\mathsf{explode}_\lor \ ts \ p) \land (\exists t \in \{\!\!\{ts\}\!\!\}.[\![\mathsf{subst}_\phi \ t \ p]\!]^{i \cdot is} = [\![\mathsf{explode}_\lor \ ts \ p]\!]^{i \cdot is}) \end{array}$

We implement $explode_{-\infty}$ (Fig. 7) and prove that it computes the right hand side of Cooper's theorem, cf. (11). It uses $all_+ d ts$ to generate all the sums of an element of $\{\!\{ts\}\!\}\$ and of some \hat{i} where $1 \leq i \leq d$, cf. (12).

$$\text{unified } p \land \{\!\!\{B\}\!\!\} = \{\!\!\{\text{bset } p\}\!\!\}$$
$$\rightarrow ([\![\exists p]\!]^{is} = [\![\text{decr}_{\phi}(\text{explode}_{-\infty} (p, B))]\!]^{is})$$
(11)

$$\exists i \in \{1..d\}. \exists b \in \{ts\}. P(\mathsf{lin}_{+} \ b \ i) = \exists t \in \{\{\mathsf{all}_{+} \ d \ ts\}. P \ t$$
(12)

Let us now look at the implementation of the decision procedure in Fig. 8. Function unify performs the U-step but also prepares the application of Cooper's theorem. For efficiency, both aset and bset are computed. Depending on their size, either the unified term and its bset or the mirrored version and its aset are passed to $explode_{-\infty}$ to compute the rhs of

$$\begin{split} & \operatorname{explode}_{-\infty} (p, B) = \\ & \operatorname{case} (\operatorname{explode}_{\vee} [\widehat{1}..\widehat{\delta_p}] p_{-}, \operatorname{explode}_{\vee} (\operatorname{all}_{+} \delta_p B) p) \text{ of } \\ & (T, _) \Rightarrow T \\ & (F, r_2) \Rightarrow r_2 \\ & (r_1, T) \Rightarrow T \\ & (r_1, F) \Rightarrow r_1 \\ & (r_1, r_2) \Rightarrow r_1 \lor r_2 \\ \\ & \operatorname{all}_{+} d [] = [] \\ & \operatorname{all}_{+} d [] \\ & \operatorname{all}_{+} d [] \\ & \operatorname{all}_{+} d [] = [] \\ & \operatorname{all}_{+} d [] \\ & \operatorname{$$

 $\operatorname{all}_{+} d (i \cdot is) = (\operatorname{map} (\operatorname{lin}_{+} i) [\widehat{1}..\widehat{d}]) @ (\operatorname{all}_{+} d is)$

Fig. 7. The rhs of Cooper's theorem

 $\begin{array}{l} \text{unify } p = \\ \textbf{let } q = \texttt{unity } p \ ; \ (A,B) = (\texttt{remdups aset } q,\texttt{remdups bset } q) \\ \textbf{in if } |B| \leq |A| \ \textbf{then } (q,B) \ \textbf{else } (\texttt{mirror } q,A) \end{array}$

 $\mathsf{cooper}\ p = \left(\lambda f.\mathsf{decr}_\phi(\mathsf{explode}_{-\infty}\ (\mathsf{unify}\ f))\right)_{+}\ (\mathsf{lin}_\phi\ (\mathsf{nnf}\ p))$

 $pa = qelim_{\phi}$ cooper

Fig. 8. The decision procedure for linearizable ϕ -formulae

Cooper's theorem. Function **cooper** composes all the normalization steps, the elimination of v_0 by unify, and the decrementation of the remaining de Bruijn indices. Function **pa** applies generic quantifier elimination to Cooper's algorithm.

Using (3), (4) and (5) we can prove

$$\begin{aligned} \operatorname{islin}_{\phi} p \wedge \operatorname{unify} p &= (q, B) \rightarrow \\ \llbracket \exists p \rrbracket^{is} &= \llbracket \exists q \rrbracket^{is} \wedge \operatorname{unified} q \wedge \{\!\!\{B\}\!\!\} = \{\!\!\{\mathsf{bset} q\}\!\!\} \end{aligned}$$
(13)

and with (11) this implies

$$\operatorname{islin}_{\phi} p \rightarrow [\![\exists p]\!]^{is} = [\![\operatorname{decr}_{\phi}(\operatorname{explode}_{-\infty}(\operatorname{unify} p))]\!]^{is}$$

which implies the correctness of cooper directly

$$\mathsf{qfree} \ q \land \mathsf{cooper} \ q = \lfloor q' \rfloor \ \to \ \mathsf{qfree} \ q' \land \llbracket \exists \ q \rrbracket^{is} = \llbracket q' \rrbracket^{is}$$

and hence, using (1), the correctness of the whole decision procedure pa:

$$\mathsf{pa} \ p = \lfloor p' \rfloor \ \to \ \llbracket p \rrbracket^{is} = \llbracket p' \rrbracket^{is} \land \mathsf{qfree} \ p'.$$

5 Formalization issues

Normal forms Cooper's decision procedure transforms the input formula into successively more specialized normal forms, which is typical for many decision procedures. In our formalization these different normal forms are specified by predicates on the input languages ϕ and ι . This has the advantage that we do not need to define new languages and translations between languages. Instead we need to add preconditions to our theorems (e.g. islin_{ι} a) and end up with more complicated function definitions (see below). Highly tuned code may require special representations of certain normal forms even using special data structures for efficiency. (e.g. [9]). For Cooper's algorithm such optimizations do not promise substantial gains.

Recursive functions The advantages of defining recursive functions by pattern matching are well known and it is used extensively in our work. Isabelle/HOL supports such definitions [19] by lists of equations. However, it is not always possible to turn each equation directly into a theorem because an equation is only applicable if all earlier equations are inapplicable. Hence Isabelle instantiates and possibly duplicates equations to make them non-overlapping. In the case of function mirror, the given list of 8 equations leads to 144 equations after disambiguation. This blowup is the result of working with the full language ϕ even when a function operates only on a certain normal form. These non-overlapping theorems are later exported to ML, which may influence the quality of the code generated by the ML compiler.

Tailored induction Isabelle/HOL derives a tailored induction rule [19] from a recursive function definition which simplifies proofs enormously. This may seem surprising since the induction rule for mirror has 144 cases. However, most of the cases are irrelevant if the argument is assumed to be linear. These irrelevant cases disappear by simplification.

6 Performance

We tested three implementations on a batch of 64 theorems, where the distribution of quantifiers is illustrated by Fig. 9. The 64 formulae contain up to five quantifiers and three quantifier alternations. The number n_q in Fig. 9 represents the number of formulae with q quantifiers. The number of quantifier alternations is also given by the n_{qi} 's. We have $n_q = n_{q0} + n_{q1} + n_{q1}$

 $n_{q2} + n_{q3}$, where n_{qi} is the number of formulae containing q quantifiers and i quantifier alternation. The column \hat{c}_{max} gives the maximal constant occurring in the given set of formulae. Finally the last column gives the speed up factor achieved.

q	n_q	n_{q0}	n_{q1}	n_{q2}	n_{q3}	\widehat{c}_{max}	speedup
1	3	3	0	0	0	24	10
2	27	20	7	0	0	13	101
3	21	2	19	0	0	129	420
4	6	1	0	0	5	6	99
5	5	3	0	5	0	12	103

Fig. 9. Number of quantifiers and speedup in the test-formulae

The adaptation of Harrison's implementation [12] (the current oracle in Isabelle/HOL) took 3.91 seconds to solve all goals. Our adaptation of this implementation to produce full proofs based on inference rules [4] took 703.08 seconds. The ML implementation obtained by Isabelle's code generator from the formally verified procedure presented above took 3.48 seconds, a speed-up of a factor of 200. All timings were carried out on a PowerBook G4 with a 1.67 GHz processor running OSX. The reason why the hand coded version is slightly slower than the generated one is that it operates on a symbolic binary representation of integers whereas the generated one uses (arbitrary precision!) ML-integers.

7 Conclusion

We presented a formally verified procedure for quantifier elimination in \mathcal{PA} . Generating ML code from it we achieved substantial performance improvements over an LCF-style implementation. Decision procedures developed this way are much easier to maintain and especially to share. Other systems supporting reflection should be able to import our work fairly directly, especially if they are of the HOL family as well.

References

- 1. Andrew W. Appel and Amy P. Felty. Dependent types ensure partial correctness of theorem provers. J. Funct. Program., 14(1):3–19, 2004.
- Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *LNCS*, pages 38–52. Springer-Verlag, 2000.

- Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In In Types for Proofs and Programs (TYPES 2000), volume 2277 of LNCS, pages 24–40. Springer-Verlag, 2002.
- A. Chaieb and T. Nipkow. Generic proof synthesis for presburger arithmetic. Technical report, TU München, 2003. http://www4.in.tum.de/~nipkow/pubs/ presburger.pdf.
- 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.
- 6. 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.
- Fischer and 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.
- 8. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Int. Conf. Functional Programming, pages 235–246. ACM Press, 2002.
- Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd, editor, *Theorem Proving in Higher Order Logics*, *TPHOLs 2005*, volume ? of *LNCS*, page ? Springer-Verlag, 2005.
- 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.
- 11. John Harrison. *Theorem proving with the real numbers*. PhD thesis, University of Cambridge, Computer Laboratory, 1996.
- John Harrison's home page. http://www.cl.cam.ac.uk/users/jrh/atp/OCaml/ cooper.ml.
- Robert Klapper and Aaron Stump. Validated Proof-Producing Decision Procedures. In C. Tinelli and S. Ranise, editors, 2nd Int. Workshop Pragmatics of Decision Procedures in Automated Reasoning, 2004.
- Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. Isabelle/HOL A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
- Michael 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 *LNCS*, pages 71–86. Springer-Verlag, 2003.
- Derek 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.
- Mojzesz 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.
- William 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.
- Konrad Slind. Derivation and use of induction schemes in higher-order logic. In TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, pages 275–290, London, UK, 1997. Springer-Verlag.