Complete Integer Decision Procedures as Derived Rules in HOL

Michael Norrish[*]

National ICT Australia
Michael.Norrish@nicta.com.au

Abstract. I describe the implementation of two complete decision procedures for integer Presburger arithmetic in the HOL theorem-proving system. The first procedure is Cooper's algorithm, the second, the Omega Test. Between them, the algorithms illustrate three different implementation techniques in a fully expansive system.

1 Introduction

Integer decision procedures are vital parts of interactive theorem-proving systems. Whether embedded in simplification routines and running automatically, or explicitly invoked by the user, they remove a great deal of tedium from the task of proving goals. Modern interactive systems, including ACL2 [10], Coq [1], HOL [4,11], Isabelle [13], Nuprl [8] and PVS [12], all implement such decision procedures.

There are essentially three procedures implemented in the systems mentioned above: Fourier-Motzkin variable elimination (in HOL, Isabelle and Coq[1]), SUP-INF [15] (in Nuprl) and Shostak's loop-residue algorithm [16] (in PVS). In this paper, I describe two other algorithms. The Omega Test [14] is an extension of Fourier-Motzkin variable elimination, which makes it complete over $\mathbb{Z}$. The second procedure is Cooper's algorithm [3], which is unlike the other algorithms in not requiring formulas to be in DNF before it eliminates a quantifier.

Both of the algorithms I describe differ in scope from the others mentioned: they are both complete over the domains covered by the others (universal Presburger arithmetic), and also complete over the wider language of Presburger formulas with any alternation of quantifiers. I will henceforth take Presburger formulas to be those generated by the grammar given in Fig. 1, and which are also *closed*: all occurrences of variables are bound by a universal or existential quantifier.

The task of the decision procedure is to prove a closed Presburger formula either valid or invalid. If a formula has existential quantifiers outermost and is proved valid, then it may be useful to have the procedure also return a satisfying assignment for the existential variables. Conversely, formulas that are universal at the outermost level, and which are proved invalid might prompt the return of a falsifying assignment. Both

[*] This work was done while supported by the Michael and Morven Heller Research Fellowship at St. Catharine's College, Cambridge.

[1] Coq implements just the first phase of the Omega Test (real shadow elimination), which is Fourier-Motzkin variable elimination.

$$\begin{array}{lcl}
\textit{formula} & ::= & \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \mid \\
 & & \neg\textit{formula} \mid \exists \textit{var}.\textit{formula} \mid \forall \textit{var}.\textit{formula} \mid \\
 & & \textit{numeral} \,|\, \textit{term} \mid \textit{term relop term} \\
\textit{term} & ::= & \textit{numeral} \mid \textit{term} + \textit{term} \mid -\textit{term} \mid \textit{numeral} * \textit{term} \mid \textit{var} \\
\textit{relop} & ::= & < \mid \leq \mid = \mid \geq \mid > \\
\textit{var} & ::= & x \mid y \mid z \ldots \\
\textit{numeral} & ::= & 0 \mid 1 \mid 2 \ldots
\end{array}$$

Fig. 1. Grammar defining Presburger formulas. Write $c|e$ to mean that c, necessarily a numeral, divides e exactly, or without remainder

algorithms proceed by quantifier elimination, eventually reducing the input formula to an equivalent formula without quantifiers.

This paper is arranged as follows: in Section 2, I describe Cooper's algorithm, including a proof of correctness. In Section 3, I describe the Omega Test, again proving correctness. In Section 4, I describe the techniques that I used to implement these procedures (proofs) within HOL. In the same section, I also briefly compare the two algorithms' performance. In Section 5, I describe extensions to the basic procedures that make them considerably more helpful in the interactive setting.

2 Cooper's Algorithm

The first step in Cooper's algorithm is to normalise the input formula. Negations are pushed inwards, so that they are only found around "divides" and equality leaves, and in front of existential quantifiers. The relations $\leq$, $\geq$ and $>$ are rewritten to forms involving $<$, and universal quantifiers are eliminated by transforming $\forall x\, P(x)$ into $\neg\exists x.\, \neg P(x)$. Note that neither normalisation to DNF nor to CNF occurs. The formula's terms are also normalised, with multiplications distributed over additions, coefficients gathered and other obvious normalisations applied.

The algorithm then arbitrarily picks an innermost (existential) quantifier to eliminate. This quantifier has scope over a formula whose tree structure has conjunctions and disjunctions at its internal nodes. The leaves of this sub-formula are now transformed to equivalent forms, where the quantifier's bound variable is isolated and has a positive coefficient. If a leaf involves the bound variable (x, say) at all, it is transformed to one of the six following forms:

$$\begin{array}{ccc} cx < e & cx = e & c|dx + e \\ e < cx & \neg(cx = e) & \neg(c|dx + e) \end{array} \tag{1}$$

where c and d are positive integer numerals, and e is an arbitrary term not including x, but possibly involving other variables.

The algorithm next finds the least common multiple (l say) of all of x's coefficients. Every leaf formula is then multiplied through by an appropriate constant so that every leaf has an occurrence of lx. The formula can then be transformed by appeal to the theorem

$$\exists x.\, P(lx) \equiv \exists x.\, P(x) \wedge l|x$$

ensuring that every occurrence of x implicitly has coefficient one.

The final phase now depends on the generation of two sets of expressions, A and B, based on the leaf expressions in the formula. Table 1 specifies how each leaf generates possible members for each set. For example, a leaf of the form $x < e$ puts e into the A-set, and does not affect the B-set. Leaves involving the divisibility relation, or which do not include the variable x, do not generate members for either set.

Leaf form	A-set	B-set
$x < e$	e	
$e < x$		e
$x = e$	$e + 1$	$e + {}^{-}1$
$\neg(x = e)$	e	e

Table 1. Generation of A and B sets

The algorithm creates two variants on the predicate P underneath the quantifier, $P_{-\infty}$ and $P_{+\infty}$. These new predicates can be understood as versions of the original where the parameter x has been made arbitrarily small (negative) or arbitrarily large, respectively. Therefore, all leaves with x free and which involve $<$ and $=$ are replaced with either true or false. For example, if x is made arbitrarily small, then $x < e$ will be true, $e < x$ will be false, $x = e$ will be false, and $\neg(x = e)$ will be true. Leaves involving divisibility will be unchanged. Finally, let δ be the least common multiple of all the c occurring in leaves of the form $c|x + e$ and $\neg(c|x + e)$.

The algorithm then eliminates the existential quantifier by using one of the two equivalences given in the following theorem. In order to reduce the amount of blow-up in term size, the first equivalence is chosen if the B-set is smaller than the A-set, and the second otherwise. If the sizes of the sets are equal, the set which has fewer occurrences of free variables is chosen.

Theorem 1 (Cooper, 1972). *Let $P(x)$ be a formula constructed of conjunctions and disjunctions of integer relations. Those relations involving x are of the form $x < e$, $e < x$, $x = e$, $x \neq e$, $c|(x + e)$ or $\neg(c|(x + e))$. Then,*

$$\exists x.\, P(x) \quad \equiv \quad \bigvee_{j=1}^{\delta} P_{-\infty}(j) \;\vee\; \bigvee_{j=1}^{\delta} \bigvee_{b \in B} P(b + j)$$

and

$$\exists x.\, P(x) \quad \equiv \quad \bigvee_{j=1}^{\delta} P_{+\infty}(j) \;\vee\; \bigvee_{j=1}^{\delta} \bigvee_{a \in A} P(a + {}^{-}j)$$

These equivalences are the heart of Cooper's algorithm. They are sufficiently similar that I will describe the proof of just the first. The equivalence is of the form $L \equiv D_1 \vee D_2$, so I prove it by showing $D_1 \Rightarrow L$, $D_2 \Rightarrow L$, and $L \wedge \neg D_2 \Rightarrow D_1$:

– The first obligation requires a proof of

$$\bigvee_{j=1}^{\delta} P_{-\infty}(j) \quad \Rightarrow \quad \exists x.\, P(x)$$

This result relies on two further lemmas. The first,

$$\exists y.\, \forall x.\, x < y \Rightarrow (P(x) \equiv P_{-\infty}(x)) \tag{2}$$

states that P and $P_{-\infty}$ coincide once their arguments are small enough. There is a witness for y; the minimum of all the expressions that occur on the other side of a $<$ or an $=$ from the bound variable in the original formula.

The second lemma is

$$\forall x\, \forall c.\, P_{-\infty}(x) \equiv P_{-\infty}(x + c\delta) \tag{3}$$

which states that the truth value of $P_{-\infty}$ is unaffected by the addition of any number of multiples of δ. Recall that the only leaves in $P_{-\infty}$ involving the bound variable x are of the form, $c|x + e$ or $\neg(c|x + e)$. As δ is the **l.c.m.** of all the c's in the formula, the truth of these leaves will be unaffected by the addition of multiples of δ to x.

The proof of the overall result follows because, from the existence of a witness satisfying $P_{-\infty}$, one can produce another, smaller than the y of (2), and thereby find a witness satisfying P.[2]

– The second obligation is

$$\bigvee_{j=1}^{\delta} \bigvee_{b \in B} P(b + j) \quad \Rightarrow \quad \exists x.\, P(x)$$

This implication follows immediately because the antecedent provides a witness that satisfies P.

– Finally, the most complicated case is

$$(\exists x.\, P(x)) \wedge \neg(\bigvee_{j=1}^{\delta} \bigvee_{b \in B} P(b + j)) \quad \Rightarrow \quad \bigvee_{j=1}^{\delta} P_{-\infty}(j) \tag{4}$$

It is sufficient to prove that

$$\neg \left(\bigvee_{j=1}^{\delta} \bigvee_{b \in B} P(b + j) \right) \Rightarrow \forall x.\, P(x) \Rightarrow P(x - \delta)$$

This is sufficient because the assumption $\exists x.\, P(x)$ from (4) provides a witness for which P is true. The new result then provides an infinite sequence of smaller witnesses, each δ smaller than the previous. In particular, there will be one that will be smaller than the y below which $P(x) \equiv P_{-\infty}(x)$ (see (2) above). Then, because $\forall x\, c.\, P_{-\infty}(x) \equiv P_{-\infty}(x + c\delta)$ (see (3)), there will be another witness for $P_{-\infty}$ that will be between 1 and δ,[3] as required.

[2] This is using another theorem about $\mathbb{Z}$, that $\forall d.\, d \neq 0 \Rightarrow \forall x\, y.\, \exists c.\, y + cd < x$.

[3] Using another lemma about $\mathbb{Z}$: $\forall d.\, 0 < d \Rightarrow \forall x.\, \exists c.\, 0 < x + cd \leq d$.

My proof is actually of the stronger statement

$$\forall P.\ Q(x) \wedge \bigwedge_{j=1}^{\delta} \bigwedge_{b \in B} \neg Q(b+j) \wedge P(x) \Rightarrow P(x-\delta) \tag{5}$$

so that an induction on the structure of P is possible. The result needed is obtained by picking Q to be P in the above.
There are two inductive steps to the proof, one for formulas constructed by conjunction and one for disjunction. Both are straight-forward: if (5) holds for P_1 and P_2, then it holds for $\lambda\, x.\ P_1(x) \wedge P_2(x)$ as well as for $\lambda\, x.\ P_1(x) \vee P_2(x)$.
There are seven leaf cases to consider, one for each of the forms in (1) and one for the case when the formula P does not mention x at all. The latter is trivially true. The other cases are as follows:

- $x < e \Rightarrow x - \delta < e$
  Follows immediately ($\delta > 0$ as it is the **l.c.m.** of positive arguments).
- $e < x \Rightarrow e < x - \delta$
  Seeking a contradiction, assume $\neg(e < x - \delta)$, i.e., $x \leq e + \delta$. So, for some $j \in 1..\delta$, $x = e + j$. The top-level assumption in (5) is $Q(x)$, so $Q(e + j)$. But e is also part of the formula's B-set, so, by assumption, $\bigwedge_{j=1}^{\delta} \neg Q(e + j)$.
- $(x = e) \Rightarrow (x - \delta = e)$
  $e + {}^{-}1$ is in the B-set, so derive a contradiction by noting that both $Q(e)$, and $\bigwedge_{j=1}^{\delta} \neg Q((e + {}^{-}1) + j)$ (pick j = 1).
- $\neg(x = e) \Rightarrow \neg(x - \delta = e)$
  Assume the opposite, so that $x = e + \delta$. Thus $Q(e + \delta)$. But e is in the B-set, so $\neg Q(e + \delta)$ (picking $j = \delta$), another contradiction.
- $c|x+e \Rightarrow c|x-\delta+e$. By construction of δ, $c|\delta$, so the result follows immediately. Similarly for the $\neg(c|x + e)$ case.

□

3 The Omega Test

The heart of the Omega Test operates on formulas of the form

$$\exists x.\ C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

where each C_i is of the form $t_1 \leq t_2$ and involves x. The procedure's initial normalisation must therefore convert input formulas to disjunctive normal form, move boolean negations and existential quantifiers inward and eliminate universal quantifiers. The $<$, $>$ and $\geq$ relations are easily converted to $\leq$ formulas. Sub-formulas $x \neq y$ are converted to $x + 1 \leq y \ \vee\ y + 1 \leq x$. The test's treatment of equality and divisibility relations is slightly more involved (see [14] for details), but is straightforward to implement.

Given a normalised formula (as above) sort the C_i into two sets, the lower bounds that are of the form $a \leq \alpha x$ with α positive, and the upper bounds that are of the form $\beta x \leq b$. If either set is empty, then immediately return true. Otherwise, if all of the coefficients of x in either set are equal to one (i.e., if all of the α_i are one, or all of the β_i are one), what Pugh refers to as *exact shadow* elimination can be performed. The original formula is equivalent to

$$\bigwedge_{i,j} a_i\beta_j \leq \alpha_i b_j \tag{6}$$

where i indexes lower bound constraints, and j the upper bounds. The formula (6) is also known as the *real shadow* because the equivalence is true, regardless of coefficients, when variables range over $\mathbb{R}$. The exact shadow result relies on the simpler equivalence

$$(\exists x.\ a \leq \alpha x \wedge \beta x \leq b) \equiv a\beta \leq \alpha b$$

when one of α or β is 1. The implication from left to right is easy to see. From right to left, with $\alpha = 1$, take x to be a; with $\beta = 1$, take x to be b. This result extends to the cross-product of many upper and lower bounds by two successive inductions on the respective sets.

If an exact shadow elimination is not possible, the relevant theorem for quantifier elimination is considerably more complicated.

Theorem 2 (Pugh, 1992). *Let $L(x)$ be a conjunction of lower bounds on x, indexed by i, of the form $a_i \leq \alpha_i x$, with α_i positive. Similarly, let $U(x)$ be a set of upper bounds on x, indexed by j, of the form $\beta_j x \leq b_j$, with β_j positive. Let m be the maximum of all the β_js. Then*

$$\begin{array}{rc}(\exists x.\ L(x) \wedge U(x)) \equiv & (\bigwedge_{i,j}(\alpha_i - 1)(\beta_j - 1) \leq \alpha_i b_j - a_i\beta_j) \\ & \vee \\ & \bigvee_i \bigvee_{k=0}^{\left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor} \exists x.\ (\alpha_i x = a_i + k) \wedge L(x) \wedge U(x)\end{array}$$

Following Pugh, the first disjunct of the RHS above is called the *dark shadow*, and the other disjuncts are called *splinters*. Note that as stated, each splinter has as many quantifiers as before. Nonetheless, the extra equality constraint means that the variable x and its quantifier can be eliminated by the normalisation techniques already mentioned, so the theorem does represent a quantifier elimination result. This theorem reduces to exact shadow elimination if all of the α_i or all the β_i are equal to 1.

The theorem is of the form $\mathit{LHS} \equiv D_1 \vee D_2$, with D_1 the dark shadow and D_2 the splinters. I prove the result by proving $D_1 \Rightarrow \mathit{LHS}$, $D_2 \Rightarrow \mathit{LHS}$ and $\mathit{LHS} \wedge \neg D_1 \Rightarrow D_2$:

- $\bigwedge_{i,j}(\alpha_i - 1)(\beta_j - 1) \leq \alpha_i b_j - a_i\beta_j \quad \Rightarrow \quad \exists x.\ L(x) \wedge U(x)$
  Show the result for one pair of upper and lower bound. I.e., that

  $$(\alpha - 1)(\beta - 1) \leq \alpha b - a\beta \ \Rightarrow\ \exists x.\ a \leq \alpha x \wedge \beta x \leq b$$

  Two inductions, on the set of lower bounds and then the upper bounds, give the complete result.
  To prove the base case, assume the opposite. Then

  $$\neg\exists x.\ a\beta \leq \alpha\beta x \leq \alpha b$$

I.e., there is no multiple of $\alpha\beta$ between $a\beta$ and αb. The other assumption implies $a\beta \leq \alpha b$ as α and β are both positive and non-zero. Take i to be the greatest multiple of $\alpha\beta$ less then $a\beta$. Then

$$\alpha\beta i < a\beta \leq \alpha b < \alpha\beta(i+1)$$

Because $0 < \alpha\beta(i+1) - \alpha b$, conclude $1 \leq \beta(i+1) - b$, and thus $\alpha \leq \alpha\beta(i+1) - \alpha b$. Similarly, $\beta \leq a\beta - \alpha\beta i$. Infer $\alpha + \beta \leq \alpha\beta + a\beta - \alpha b$, or (re-arranging), $\alpha b - a\beta < \alpha\beta - \alpha - \beta + 1$, which contradicts the first assumption.

– $\bigvee_i \bigvee_{k=0}^{\left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor} \exists x.\ (\alpha_i x = a_i + k) \land L(x) \land U(x) \quad \Rightarrow \quad \exists x.\ L(x) \land U(x)$
  Trivial, as each splinter disjunct on the left provides an x that will satisfy the weaker requirement on the right.
– $(\exists x.\ L(x) \land U(x)) \land \neg(\bigwedge_{i,j} (\alpha_i - 1)(\beta_j - 1) \leq \alpha_i b_j - a_i \beta_j) \quad \Rightarrow$
  $\bigvee_i \bigvee_{k=0}^{\left\lfloor \frac{m\alpha_i - \alpha_i - m}{m} \right\rfloor} \exists x.\ (\alpha_i x = a_i + k) \land L(x) \land U(x)$
  Let x be the witness to the first assumption. The second assumption means that there exist α, β, a and b such that

$$\alpha b - a\beta \leq \alpha\beta - \beta - \alpha \tag{7}$$

These values occur in constraints from L and U, so $\beta x \leq b$ and $a \leq \alpha x$. Multiplying the former through by α gives $\alpha\beta x \leq \alpha b$, so in conjunction with (7)

$$\begin{aligned} &\alpha\beta x && \leq a\beta + \alpha\beta - \beta - \alpha \\ \Rightarrow\ &\beta(\alpha x - a) && \leq \alpha\beta - \beta - \alpha \\ \Rightarrow\ &\alpha x - a && \leq \left\lfloor \frac{\alpha\beta - \beta - \alpha}{\beta} \right\rfloor \end{aligned}$$

All of the β coefficients are $\leq m$, so

$$\left\lfloor \frac{\alpha\beta - \beta - \alpha}{\beta} \right\rfloor \leq \left\lfloor \frac{m\alpha - \alpha - m}{m} \right\rfloor$$

There is now enough information to pick the appropriate disjunct from the RHS. The α_i is α and k is $\alpha x - a$.

□

Even when exact shadow elimination is not possible, it is still worth checking the real shadow of a formula if all of its variables are bound by existential quantifiers. The latter condition means that performing all of the real shadow eliminations will result in a ground formula. If this formula is false, then so too is the original formula. Thus [14] describes the algorithm for the purely existential case in three stages:

1. Check the real shadow. If it is unsatisfiable, so is the original. If it is satisfiable and the shadow is exact, then the original formula is satisfiable.
2. Check the dark shadow. If it is satisfiable, so is the original.
3. Check the splinters.

4 Implementations in HOL

HOL is a theorem-proving system in the tradition of LCF [5]. Theorems are implemented as an abstract data type in the language SML, exploiting that language's strong typing system, which guarantees that a type's abstraction barrier can not be subverted. The implementation of the type of theorems provides functions, known as the *primitive rules of inference*, for manipulating and creating theorem values. Strong typing then ensures that all such values must ultimately be constructed using just these rules. Rich suites of tools for proving theorems are built on top of this *logical kernel*, but all proofs are *fully expanded* to sequences of primitive inferences. No tool or derived rule has the ability to simply assert a theorem; proofs must be provided.[4]

The implementations of Cooper's algorithm and the Omega Test in HOL illustrate three different techniques for realising decision procedures in a fully expansive setting. Before examining each technique's use in the implementation of the two decision procedures, I will describe each in general terms, as well as giving examples of their use in other applications.

Theorem instance re-proof: This, the simplest but also most naïve technique, plays out the central proof of a decision procedure's core theorem(s) for every problem instance. This approach is used by the implementation of Cooper's algorithm.

For a contrived example, consider a proof procedure to turn a theorem of the form $\vdash P \Rightarrow Q \Rightarrow R$ into a theorem of the form $\vdash P \land Q \Rightarrow R$. The ML code implementing this transformation would involve calls to the inference rules given as labels in this inference tree:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{P \land Q \vdash P \land Q}\;\texttt{ASSUME}}{P \land Q \vdash Q}\;\texttt{CONJ2} \qquad \cfrac{\cfrac{\cfrac{}{P \land Q \vdash P \land Q}\;\texttt{ASSUME}}{P \land Q \vdash P}\;\texttt{CONJ1} \qquad \cfrac{\cfrac{\vdash P \Rightarrow Q \Rightarrow R}{P \vdash Q \Rightarrow R}\;\texttt{UNDISCH}}{P, Q \vdash R}\;\texttt{UNDISCH}}{P \land Q, Q \vdash R}\;\texttt{PROVE\_HYP}}{P \land Q \vdash R}\;\texttt{PROVE\_HYP}}{\vdash P \land Q \Rightarrow R}\;\texttt{DISCH}$$

This procedure will run in constant time in terms of number of primitive inferences (nine in this case), but is clearly awkward and inefficient. It is as well that all proof programming in HOL doesn't have to drop to this level. The *pro forma* theorem approach described below is much more appropriate here: prove the equivalence

$$\vdash (P \Rightarrow Q \Rightarrow R) \equiv (P \land Q \Rightarrow R)$$

once and then instantiate this theorem as and when necessary.

Theorem instance re-proof is not always so obviously inappropriate. It is particularly useful when it is impossible or difficult to express the generalised theorem in HOL's logic. For example, in a suitable logic about polytypic functions, the following might be true

$$map\ f \circ map\ g = map(f \circ g)$$

[4] HOL does not have actual "proof objects". At most, a proof can be regarded a sequence of calls to the kernel's API; only the theorem it creates has any lasting existence.

where map was a polytypic function that mapped functions over the values in appropriate container types, without disturbing the structure of the container. HOL's logic can not give a type to or define such a map. Nevertheless, given suitable definitions of the types and their map functions, it is not difficult to write a proof procedure that automatically proves the instances (over lists and trees, for example) of the more general "theorem".

Using *pro forma* theorems: As has already been suggested, this approach involves proving suitably general theorems embodying the core of the decision procedure (often direct equivalences between formulas). These theorems can then be instantiated with specific problems, quickly implementing one or more steps of the decision procedure. The "symbolic" implementation of the Omega Test (described below) uses this technique.

One problem with this technique arises when the theorem in question relates its LHS to some new formula on the right that is a function of the actual form of the LHS. Theorems 1 and 2 demonstrate this problem: Cooper's theorem requires the construction of the new $P_{-\infty}$ and $P_{+\infty}$ formulas; Pugh's theorem requires the generation of the pair-wise product of a formula's upper and lower-bound constraints. The example of the propositional rewrite above ($(P \Rightarrow Q \Rightarrow R) \equiv (P \wedge Q \Rightarrow R)$) doesn't have this problem because the equivalence doesn't require any action on the formulas instantiated for P, Q and R.

If, for example, P is a predicate on $\mathbb{Z}$, with HOL type `int -> bool`, it is difficult to directly define a function to calculate $P_{-\infty}$.[5] Instead, Harrison's "shadow syntax" approach can be used [6]: a concrete type is used to implement a syntax for the formula, and an interpretation function relates this new syntax back to the original domain. Thus, in his implementation of Kreisel and Krivine's quantifier elimination algorithm for the elementary first order theory of $\mathbb{R}$, Harrison uses a constant `poly` with defining equations[6]

```
poly x [] = 0
poly x (h::t) = h + x * poly x t
```

In this way, a polynomial on variable x can be reduced to `poly` x followed by a list of the coefficients of the powers of x. Similarly, conjunctions and disjunctions of relations on polynomials are represented as lists of polynomials, interpreted by appropriate functions. Harrison proves the theorems that form the basis for the algorithm, with those theorems' manipulations of the syntax of the formulas represented by manipulations of the various lists that make up the shadow syntax.

One further advantage of the use of *pro forma* theorems is that they can provide an elegant packaging of an algorithm's fundamentals as data (the theorems themselves), rather than as the code necessary in the re-proof technique above.

External proof discovery: This technique is appropriate in applications where a decision procedure does most of its work finding a proof, and where the proof itself, however construed, is relatively small. The core idea is to do proof discovery outside

[5] One issue is that the size of the domain is uncountable, but the desired function only operates over the countable subset corresponding to syntactically expressible formulas.

[6] A note on HOL syntax: `[]` is the empty list; `h::t` is the list consisting of element `h` "cons"-ed onto the list `t`.

of the logical kernel, so that the kernel's general-purpose machinery can be replaced with special-purpose code tuned to the particular application. It is only when the proof is found that the kernel becomes involved; it executes the proof itself and confirms that the special-purpose code was correct.[7]

This technique is clearly applicable in deciding first order logic. There, proofs are typically very short in comparison to the work done in exploring all of the possible paths to a negated goal's refutation. This approach is exemplified by Hurd's linking of HOL to the Gandalf resolution prover [7]. There the external tool is external not just to the logical kernel, but to HOL itself. On proving a goal, Gandalf provides a log of the successful resolution and modulation steps required, and Hurd's HOL code then interprets this proof-log "back into" the logical kernel.

Another example of the technique is Boulton's implementation of his procedure for universal Presburger arithmetic on $\mathbb{N}$ [2]. On this domain, Fourier-Motzkin variable elimination can be seen as a refutation procedure. Boulton translates the negated goal into a special data structure representing the set of known constraints. When new consequences are inferred, each is accompanied by a closure that, when run, would prove that consequence.[8] If false is inferred, then just those inferences leading to that conclusion need to be replayed in the logical kernel. This implementation inspired the "no alternating quantifiers" part of the Omega Test.

4.1 Implementing Cooper's Algorithm

The implementation of Cooper's algorithm uses the theorem instance re-proof technique. Though it uses some pre-proved theorems (the various lemmas about $\mathbb{Z}$ mentioned in the course of the proof, for example), the bulk of the proof is replayed for every proof instance. The calculation of the RHS of Theorem 1, particularly the predicate $P_{-\infty}$ (or $P_{+\infty}$ depending on the choice of equivalence), can be done extra-logically, but the required properties of the new formulas on the right must still be proved.

Although the presented proof involves an induction over the structure of the general formula P, the implementation only ever proves a concrete instance (P is given as an input), so there is no induction. Instead, the procedure for proving the instance is written recursively. The procedure starts with the theorem $P(x) \vdash P(x)$. If the conclusion is of the form $P_1(x) \wedge P_2(x)$, the procedure makes recursive calls on $P(x) \vdash P_1(x)$ and $P(x) \vdash P_2(x)$ to prove $P(x) \vdash P_1(x - \delta)$ and $P(x) \vdash P_2(x - \delta)$ (assuming the B-set is being used). It is easy to then combine these theorems to generate

$$P(x) \vdash P_1(x - \delta) \wedge P_2(x - \delta)$$

Disjunctions are handled similarly. At the top-level the required implication is generated by discharging the assumption. The use of $P(x)$ as an unchanging assumption throughout the recursion models the use of the predicate Q in the earlier proof of Theorem 1.

I also implemented a version of the algorithm using a *pro forma* theorem. This required the proof in HOL of the general statement of the theorem. The shadow syntax was defined in the logic as a free algebra, with constructors such as `Conjn`, `Disjn` and

[7] This technique can also be seen as certificate checking; see [6, §6] for an extended discussion.

[8] The use of closures in this case is a specific instance of Boulton's general idea of *lazy theorems*.

xLT. This type provided concrete syntax for predicates over integers. The function for evaluating shadow syntax was `eval_form`, taking a formula in this concrete form and a value to evaluate with respect to it. Thus, `eval_form` can be seen as an implementation of the action of applying a predicate to an argument. Its definition was

```
eval_form (Conjn f1 f2) x = eval_form f1 x ∧ eval_form f2 x
eval_form (Disjn f1 f2) x = eval_form f1 x ∨ eval_form f2 x
eval_form (Negn f) x = ¬eval_form f x
eval_form (UnrelatedBool b) x = b
eval_form (xLT i) x = x < i
eval_form (LTx i) x = i < x
eval_form (xEQ i) x = (x = i)
eval_form (xDivided i1 i2) x = i1 int_divides x + i2
```

The final theorem proved in HOL was of the form

```
⟨side conditions⟩ ⇒ ((∃x. eval_form f x) ≡ ...)
```

This "shadow syntax" implementation of the core part of the algorithm, coded as a complete replacement for the theorem instance re-proof component, performed slightly worse than the original (though only on a small regression test-suite, see below). With the *pro forma* approach, the final step of instantiating the core theorem is constant time, but this must be preceded by an $O(n)$ translation of the formula into its shadow syntax, and a similar translation back out afterwards. Both approaches should thus have the same asymptotic complexity, and determining which to use in practice requires experimentation.

The HOL implementation also includes most of Cooper's other suggested improvements to his algorithm, including those designed to prevent or ameliorate the "δ-expansion". Though it seems impossible to prevent the introduction of new disjunctions over elements of the A or B-sets, one can delay having to expand the disjunctions over the $j \in 1 \dots \delta$ until all of the quantifiers have been eliminated. If the final formula then includes divisibility constraints on the variables j, then the range of these constraints can be reduced, resulting in fewer disjuncts to check.

4.2 Implementing the Omega Test

The implementation of the Omega Test consists of two loosely coupled components. One uses external proof discovery, and finds refutations or satisfying assignments for existential goals. The other, "symbolic" sub-system uses *pro forma* theorems to perform quantifier elimination on formulas with alternating quantifiers, and also on existential formulas that produce "splinters".

After an input formula is normalised, it is passed to the appropriate sub-system. Both can result in the other being called. If an existential goal is not exact, is not refuted by its real shadow, and has no satisfying assignment found in its dark shadow, then it is passed to the symbolic sub-system, which will eliminate one quantifier. When the symbolic sub-system eliminates a quantifier, the resulting formula may be purely universal or existential. If so, it is passed to the external proof discovery sub-system.

The external proof discovery sub-system is inspired by Boulton's existing code for $\mathbb{N}$ in HOL. There are two interesting differences between that implementation and mine. Where his code uses closures to represent possible proofs, I use an explicit, concrete ML data structure. Its declaration is

```
datatype 'a derivation =
  ASM of 'a
| REAL_COMBIN of int * 'a derivation * 'a derivation
| GCD_CHECK of 'a derivation
| DIRECT_CONTR of 'a derivation * 'a derivation
```

A `derivation` represents the proof of a formula

$$0 \leq c_1 v_1 + \ldots + c_n v_n + c$$

where the v_i are the existentially bound variables. The four constructors in the type correspond to making an assumption, combining a lower and upper bound at variable i, reducing a constraint because all of its variable coefficients have a common divisor, and finding a direct contradiction, where one constraint is of the form $0 \leq X + c$, the other is $0 \leq {}^{-}X + d$ and where ${}^{-}c \not\leq d$. (X can be the sum of any number of $c_i v_i$ pairs, allowing a refutation to be found before all variables are eliminated.)

In practice, the polymorphic `'a` parameter of the `derivation` type is instantiated to `term`, the type of HOL terms. Nonetheless, the implementation of the external proof discovery analysis doesn't require any connection with the HOL kernel and has been used independently by others. This is in contrast with Boulton's code, which is tied to the HOL kernel because its closures are of pending calls to the kernel's rules of inference.

More significantly, Boulton's code only finds refutations. (An implementation of SUP-INF is used separately to find witnesses for existential goals.) This is because a goal is not necessarily satisfiable if its real shadow does not refute it. On the other hand, the Omega Test implementation can find satisfying assignments when it reduces an exact or dark shadow to a formula of just one variable. Such a formula has at most two constraints, a lower and upper bound. A coefficient that isn't 1 or ${}^{-}1$ can be divided out, and one of $0 \leq cx + c_1$ and $0 \leq cx + c_2$ will imply the other, because c_1 and c_2 are constants.[9] If the constraints aren't contradictory, then either bound will satisfy the formula. Recursively unwinding the computation of the shadow, satisfying assignments can be found for other variables. The original goal can then be proved as a HOL theorem using the provided witnesses.

The "symbolic" sub-system is based on constants `evallower` and `evalupper`, which are used to interpret lists of pairs of numbers as lower and upper-bound constraints on a provided value. The defining equations for `evallower` are

```
evallower x [] = ⊤
evallower x ((c,lb)::cs) = lb <= &c * x ∧ evallower x cs
```

[9] Following [14], my implementation uses a hash-table to efficiently eliminate redundant constraints.

(The first number of each pair is a natural number, ensuring that the bound really is a lower-bound, so the & function is used to inject from $\mathbb{N}$ to $\mathbb{Z}$. The `evalupper` function is defined similarly.)

The theorem representing exact shadow elimination is simple to state:

```
EVERY fst_nzero uppers ∧ EVERY fst_nzero lowers ⇒
EVERY fst1 uppers ∨ EVERY fst1 lowers ⇒
((∃x. evalupper x uppers ∧ evallower x lowers) ≡
 real_shadow uppers lowers)
```

The side-conditions require that the coefficients are all non-zero, and that either all of the upper or lower coefficients are equal to 1. The `real_shadow` function constructs the new set of constraints. It is characterised thus:

```
real_shadow uppers lowers =
  ∀c d lb ub.
    MEM (c,ub) uppers ∧ MEM (d,lb) lowers ⇒
    &c * lb <= &d * ub
```

Finally, the HOL theorem corresponding to Theorem 2:

```
EVERY fst_nzero uppers ∧ EVERY fst_nzero lowers ∧
EVERY (λp. FST p <= m) uppers ⇒
((∃x. evalupper x uppers ∧ evallower x lowers) ≡
 dark_shadow uppers lowers ∨ ∃x. splinter x m uppers lowers)
```

This restatement of the theorem uses the fact that the variable `m` need not be the maximum of the upper bound coefficients, but needs only be at least as big as them all. The implementation always instantiates `m` to be the maximum, but it's simpler not to have to compute the maximum in the logic.

4.3 Comparing the Algorithms

I have not performed extensive performance analyses of the implementations of the algorithms. Both were developed and tested against the same regression test-suite, which, now that the algorithms' implementations are complete, contains 152 problems. Though a small collection of mainly small problems, it is a plausible test of the algorithms over one of the domains where they're most likely to be applied: interactive proof of goals that are usually valid. Another similar domain is the use of these algorithms as part of simplification. There, though the Presburger goals are more likely to be invalid, they are still usually small parts of larger goals.

The figures, based on one run of each algorithm over the suite, reveal that the Omega Test is 22% faster over the whole test-suite (on a dual processor 1.6GHz Athlon machine, Omega did the 152 problems in 54.7s, Cooper's algorithm in 66.8s). Omega solved the problem with the biggest absolute time difference in 5.4s, and Cooper's algorithm in 9.0s, so this one problem accounts for roughly a third of the difference between them. Cooper's algorithm is faster on 59 of the problems (on 16, the algorithms take equally long). Cooper's algorithm's best performance, in terms of absolute time elapsed, is a problem where it takes 4.2s, and the Omega Test takes 5.8s.

These figures can only be used to indicate that one implementation of the Omega Test seems a little quicker than one implementation of Cooper's algorithm, and that there are input problems for each where one is slower than the other. The implementation of the Omega Test has the advantage that it can work outside the logic when solving purely existential goals. Conversely, Cooper's algorithm need not convert to DNF. It is simple to construct problems to favour one or other of these strengths.

Experiments to compare the algorithms more rigorously would need careful construction. In recent work, Janičić, Green and Bundy compared implementations of Cooper's algorithm and Hodes's method applied to universal natural number Presburger goals [9]. The tests used for this work were generated randomly, and suggested that Cooper's algorithm could perform as well as Hodes's method.

5 Extensions

After implementing the basic algorithms, it is possible to extend their "reach" by applying a variety of pre-processing steps to their input formulas. Here are descriptions of some of these:

Generalisation. Where a formula contains free variables, or other terms involving symbols outside Presburger arithmetic, attempt to prove the goal where these terms are replaced with a universally quantified variable. This enhancement is trivial, but makes a big difference to usability in an interactive setting. When the goal is as likely to be unsatisfiable as valid, such as when the procedure is embedded in a simplifier, it also makes sense to negate the input formula, generalise and then try to prove the resulting goal.

Natural numbers. Closed formulas where quantified variables range over $\mathbb{N}$ can be easily converted to equivalent formulas whose quantified variables range over $\mathbb{Z}$. This makes the one procedure universal for both domains, and their mixture. The equations used for the quantifiers are

$$\begin{aligned}(\exists\, x : \mathbb{N}.\; P(\&x)) &\equiv\; (\exists\, x : \mathbb{Z}.\; 0 \leq x \wedge P(x))\\ (\forall\, x : \mathbb{N}.\; P(\&x)) &\equiv (\forall\, x : \mathbb{Z}.\; 0 \leq x \Rightarrow P(x))\end{aligned}$$

where $\&$ is the function which injects from $\mathbb{N}$ into $\mathbb{Z}$. Conversion of predicates so that they range over $\mathbb{Z}$ instead of $\mathbb{N}$ is done using equations such as

$$\begin{aligned}n <_{\mathbb{N}} m \quad &\equiv \&n <_{\mathbb{Z}} \&m\\ \&(n +_{\mathbb{N}} m) &= \&n +_{\mathbb{Z}} \&m\end{aligned}$$

Dealing with natural numbers also requires a separate phase of generalisation. For example, the valid formula $0 \leq n{*}m$ with $n, m \in \mathbb{N}$ must be turned into $\forall p : \mathbb{N}.\; 0 \leq p$ and then to $\forall i : \mathbb{Z}.\; 0 \leq i \Rightarrow 0 \leq i$. Omitting this first phase of generalisation will result in first, the non-Presburger $\forall i\, j : \mathbb{Z}.\; 0 \leq i * j$, and then $\forall k : \mathbb{Z}.\; 0 \leq k$, both of which are invalid.

Expansion of constants. It is a trivial matter to expand many useful constants that will likely appear in goals, replacing occurrences with their definitions or appropriate

characterising theorems. For example, predicates such as ODD, the unique existence quantifier ($\exists!$) and functions such as max can all be treated directly.
It is even possible to get a treatment of integer division (as long as the divisor is a non-zero constant) by observing that

$$P(x/d) \equiv \exists q\, r.\, (x = qd + r) \land (0 \leq r < d \lor d < r \leq 0) \land P(q)$$

The same technique also provides a treatment of modulus.

6 Conclusions

I have presented detailed proofs of the correctness of two complete decision procedures for full Presburger integer arithmetic. On this foundation, I described how these procedures (and in some sense, these proofs) have been implemented in the HOL theorem-proving system. The implementations further demonstrate three important techniques with which complicated algorithms can be realised in a fully expansive setting.

As a demonstration of what is possible, this work supports the call in [9] for implementors of theorem-proving systems to provide complete methods. In particular, because the Omega Test will prove all of those universal goals that incomplete implementations of Fourier-Motzkin variable elimination prove, using the same approach, there seems little reason not to extend these implementations to also prove the other universal goals, and to then also cope with alternating quantifiers.

The sketchy testing I have performed seems to indicate that "one can never have too many decision procedures". If one encounters a goal that is not handled well by one algorithm, it is useful to have another weapon in one's armoury. This holds for interactive tool use, where the user makes the selection, and also for automatic systems that combine algorithms, perhaps by running them in parallel.

Availability. All of the source code (and the test-suite) for the implementations is part of the standard HOL distribution, available from SourceForge, at `http://hol.sourceforge.net`.

References

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. See the Coq home-page at `http://coq.inria.fr/`.

[2] R. J. Boulton. *Efficiency in a fully-expansive theorem prover.* PhD thesis, Computer Laboratory, University of Cambridge, May 1994.

[3] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.

[4] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[5] M. J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Number 78 in Lecture Notes in Computer Science. Springer, 1979.

[6] John Harrison. *Theorem Proving with the Real Numbers.* CPHC/BCS Distinguished Dissertations. Springer, 1998.

[7] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, Berlin, September 1999. Springer.

[8] Paul B. Jackson. *The Nuprl Proof Development System. Version 4.1 Reference Manual and User's Guide.* Cornell University, Ithaca, 1994. See the Nuprl home-page at `http://www.cs.cornell.edu/Info/Projects/NuPrl/html/NuprlSystem.html`.

[9] P. Janičić, I. Green, and A. Bundy. A comparison of decision procedures in Presburger arithmetic. In *Proceedings of LIRA'97: the 8th Conference on Logic and Computer Science*, pages 91–101. University of Novi Sad, 1997. Also available as Research Paper 872, Department of AI, University of Edinburgh.

[10] Matt Kaufmann and J Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[11] Michael Norrish and Konrad Slind. A thread of HOL development. *Computer Journal*, 45(1):37–45, 2002.

[12] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[13] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.

[14] William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[15] R. E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[16] R. E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.