A Proof-Producing Decision Procedure for Real Arithmetic

Sean McLaughlin[1] and John Harrison[2]

[1] Computer Science Department, Carnegie-Mellon University
5000 Forbes Avenue, Pittsburgh PA 15213, USA
seanmcl@cmu.edu
[2] Intel Corporation, JF1-13
Hillsboro OR 97124, USA
johnh@ichips.intel.com

Abstract. We present a fully proof-producing implementation of a quantifier elimination procedure for real closed fields. To our knowledge, this is the first generally useful proof-producing implementation of such an algorithm. While many problems within the domain are intractable, we demonstrate convincing examples of its value in interactive theorem proving.

1 Overview and related work

Arguably the first automated theorem prover ever written was for a theory of linear arithmetic [8]. Nowadays many theorem proving systems, even those normally classified as 'interactive' rather than 'automatic', contain procedures to automate routine arithmetical reasoning over some of the supported number systems like $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$. Experience shows that such automated support is invaluable in relieving users of what would otherwise be tedious low-level proofs. We can identify several very common limitations of such procedures:

- Often they are restricted to proving purely universal formulas rather than dealing with arbitrary quantifier structure and performing general quantifier elimination.
- Often they are not complete even for the supported class of formulas; in particular procedures for the integers often fail on problems that depend inherently on divisibility properties (e.g. $\forall x\ y \in \mathbb{Z}.\ 2x + 1 \neq 2y$)
- They seldom handle non-trivial nonlinear reasoning, even in such simple cases as $\forall x\ y \in \mathbb{R}.\ x > 0 \wedge y > 0 \Rightarrow xy > 0$, and those that do [18] tend to use heuristics rather than systematic complete methods.
- Many of the procedures are standalone decision algorithms that produce no certificate of correctness and do not produce a 'proof' in the usual sense. The earliest serious exception is described in [4].

Many of these restrictions are not so important in practice, since subproblems arising in interactive proof can still often be handled effectively. Indeed, sometimes the restrictions are unavoidable: Tarski's theorem on the undefinability of truth implies that there cannot even be a complete semidecision procedure for nonlinear reasoning over

the integers. At the other end of the tower of number systems, one of the few implementations that has none of the above restrictions is described in [16], but that is for the complex numbers where quantifier elimination is particularly easy.

Over the real numbers, there are algorithms that can in principle perform quantifier elimination from arbitrary first-order formulas built up using addition, multiplication and the usual equality and inequality predicates. In this paper we describe the implementation of such a procedure in the HOL Light theorem prover [14], a recent incarnation of HOL [11]. It is in principle complete, and can handle arbitrary quantifier structure and nonlinear reasoning. For example it is able to prove the criterion for a quadratic equation to have a real root automatically:

$$\forall a\ b\ c.\ (\exists x.\ ax^2 + bx + c = 0) \Leftrightarrow a = 0 \wedge (b = 0 \Rightarrow c = 0) \vee a \neq 0 \wedge b^2 \geq 4ac$$

Similar — and indeed more powerful — algorithms have been implemented before, the first apparently being by Collins [7]. However, our algorithm has the special feature that it is integrated into the HOL Light prover and rather than merely asserting the answer it *proves* it from logical first principles.

The second author has previously implemented another algorithm for this subset in proof-producing style [15] but the algorithm was so inefficient that it never managed to eliminate two nested quantifiers and has not been useful in practice. The closest previous work is by Mahboubi and Pottier in Coq [21], who implemented precisely the same algorithm as us — in fact we originally learned of the algorithm itself via Pottier. However, while it appeared to reach a reasonable stage of development, this procedure seems to have been abandoned and there is no version of it for the latest Coq release. Therefore, our algorithm promises to be the first generally useful version that produces proofs.

2 Theoretical background

In this section we describe the theoretical background in more detail. Some of this material will already be familiar to the reader.

2.1 Quantifier elimination

We say that a theory T in a first-order language L *admits quantifier elimination* if for each formula p of L, there is a quantifier-free formula q such that $T \models p \Leftrightarrow q$. (We assume that the equivalent formula contains no new free variables.) For example, the well-known criterion for a quadratic equation to have a (real) root can be considered as an example of quantifier elimination in a suitable theory T of reals:

$$T \models (\exists x.\ ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geq 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$$

If a theory admits quantifier elimination, then in particular any closed formula (one with no free variables, such as $\forall x.\ \exists y.\ x < y$) has a T-equivalent that is ground, i.e.

contains no variables at all. In many cases of interest, we can quite trivially decide whether a ground formula is true or false, since it just amounts to evaluating a Boolean combination of arithmetic operations applied to constants, e.g. $2 < 3 \Rightarrow 4^2 + 5 < 23$. (One interesting exception is the theory of algebraically closed fields of unspecified characteristic, where quantifiers can be eliminated but the ground formulas cannot in general be evaluated without knowledge about the characteristic.) Consequently quantifier elimination in such cases yields a decision procedure, and also shows that such a theory T is complete, i.e. every closed formula can be proved or refuted from T. For a good discussion of quantifier elimination and many explicit examples, see [19].

2.2 Real-closed fields

We consider a decision procedure for the theory of real arithmetic with addition and multiplication. While we will mainly be interested in the real numbers $\mathbb{R}$, the same procedure can be exploited for more general algebraic structures, so-called *real closed fields*. The real numbers are characterized up to isomorphism by the axioms for an ordered field together with some suitable second-order completeness axiom (e.g. 'every bounded nonempty set of reals has a least upper bound'). The real-closed field axioms are those for an ordered field together with the assumptions that every nonnegative element has a square root:

$$\forall x.\ x \geq 0 \Rightarrow \exists y.\ x = y^2$$

and second that all polynomials of odd degree have a root, i.e. we have an infinite set of axioms, one like the following for each *odd* n:

$$\forall a_0, \ldots, a_n.\ a_n \neq 0 \Rightarrow \exists x.\ a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0.$$

We will implement quantifier elimination for the reals using quite a number of analytic properties. All of these have been rigorously proven in HOL Light starting from a definitional construction of the reals [15]. However, these proofs sometimes rely on the completeness property, which is true for the reals but not for all real-closed fields. With more work, we could in fact show that all these analytic facts follow from the real-closed field axioms alone, and hence make the procedure applicable to other real-closed fields (e.g. the algebraic or computable reals). However, since we don't envisage any practical applications, this is not a high priority.

2.3 Quantifier elimination for the reals

A decision procedure for the theory of real closed fields, based on quantifier elimination, was first demonstrated by Tarski [30][1]. However, Tarski's procedure, a generalization of the classical technique due to Sturm [29] for finding the number of real roots of a univariate polynomial, was both difficult to understand and highly inefficient in

[1] Tarski actually discovered the procedure in 1930, but it remained unpublished for many years afterwards.

practice. Many alternative decision methods were subsequently proposed; two that are significantly simpler were given by Seidenberg[27] and Cohen[6].

Perhaps the most efficient general algorithm currently known, and the first actually to be implemented on a computer, is the Cylindrical Algebraic Decomposition (CAD) method introduced by Collins[7]. A relatively simple but rather inefficient, algorithm is also given in [19] (see [9] for a more leisurely description). Another even simpler but generally rather more efficient algorithm is given by Hörmander [17] based on an unpublished manuscript by Paul Cohen[2] (see also [10, 3] and a closely related algorithm due to Muchnik [26, 22]). It was this algorithm that we chose to implement.

2.4 Fully-expansive decision procedures

Theorem provers like HOL Light belong to the tradition established in Edinburgh LCF [12], where all theorems must be produced by application of simple primitive logical rules, though arbitrary programmability can be used to compose them. Thus, we need a procedure that does not simply assert that a formula is a quantifier-free equivalent of the input, but *proves* it from first principles.

At first sight, implementing decision procedures such that they produce proofs seems a daunting task. Indeed, it is in general significantly harder than simply writing a standalone 'black box' that returns an answer. However, if we want to really be sure about correctness, the only other obvious alternative, often loosely called 'reflection' [13], is to formally prove a standalone implementation correct. This is generally far more difficult again, and has so far only been applied to relatively simple algorithms. Moreover, it is of no help if one wants an independently checkable proof for other reasons, e.g. for use in proof-carrying code [23].

Even discounting the greater implementation difficulty of a proof-producing decision procedure, what about the cost in efficiency of producing a proof? In many cases of practical interest, neither the implementation difficulty nor the inefficiency need be as bad as it might first appear, because it is easy to arrange for a more computationally intensive phase not so different from a standalone implementation to produce some kind of certificate that can then be checked by the theorem prover. Since inference only needs to enter into the second phase, the overall slowdown is not so large. The first convincing example seems to have been [20], where a pretty standard first-order prover is used to search for a proof, which when eventually found, is translated into HOL inferences. Blum [2] generalizes such observations beyond the realm of theorem proving, by observing that in many situations, having an algorithm produce an easily checkable certificate is an effective way of ensuring result correctness, and more tractable than proving the original program correct.

In a more 'arithmetical' vein, the second author has recently been experimenting with a technique based on real Nullstellensatz certificates to deal with the universal subset of the present theory of reals [24]. This involves a computationally expensive search using a separate semidefinite programming package, but this search usually results in a compact certificate which needs only a few straightforward inferences to ver-

[2] 'A simple proof of Tarski's theorem on elementary algebra', mimeographed manuscript, Stanford University 1967.

ify. For example, using this procedure we can verify the 'universal half' of the quadratic example:

$$\forall a\ b\ c\ x.\ ax^2 + bx + c = 0 \Rightarrow b^2 - 4ac \geq 0$$

by considering the certificate:

$$b^2 - 4ac = (2ax + b)^2 - 4a(ax^2 + bx + c)$$

Since the first term on the right is a square, and the second is zero by hypothesis, it is clear that the LHS is nonnegative. Almost all the computational cost is in coming up with the appropriate square term and multiple of the input equation to make this identity hold; checking it is then easy.

However, Hörmander's algorithm (in common with all others for the full theory that we are familiar with) does not seem to lend itself to this kind of separation of 'search' and 'checking', and so we need to essentially implement all the steps of the procedure in a theorem-producing way. However, we can make this somewhat more efficient, as well as more intellectually manageable, by proving very general lemmas that apply to large families of special cases. By coding up relatively complicated syntactic structures using logical constructs, we avoid re-proving many analytical lemmas for many different cases. This will be seen more clearly when we look at the implementation of the algorithm in detail.

3 The Algorithm

Our procedure was designed by systematically modifying to produce proofs a standalone implementation of Hörmander's algorithm in OCaml.[3] We will sometimes explain the algorithm with reference to this code, since it shows the detailed control flow explicitly; it is hoped that this will still be clarifying even though it sometimes contains other functions that are not explained. In the next section we consider some of the special problems that arise when reimplementing the procedure in proof-producing style. It is instructive to see the parallels and differences: the basic control flow is all but identical, yet we produce theorems at each stage, replacing ad hoc term manipulation by logical inference.

Note first that, since our terms are built up from constants by negation, addition, subtraction and multiplication, we can rewrite all the atomic formulas in the form $p(x_1, \ldots, x_n) \bowtie 0$ where $p(x_1, \ldots, x_n)$ is a polynomial in $x_1, \ldots, x_n$ and $\bowtie$ is an equality or inequality predicate ($=$, $\leq$, $<$, $\neq$ etc.) It greatly helps if we initially rewrite the polynomials into a canonical representation and maintain this throughout the algorithm. In particular, we regard a multivariate polynomial $p(x_1, \ldots, x_n)$ as a polynomial in x_n with parameters polynomials in $p(x_1, \ldots, x_{n-1})$, each of those in turn regarded as a polynomial in x_{n-1} etc., where the sorting of the variables is determined by the nesting of quantifiers, x_1 being the outermost and x_n the innermost.

[3] Available from `http://www.cl.cam.ac.uk/users/jrh/atp` in `real.ml` with some support functions from other files.

3.1 The role of sign matrices

The key idea of the algorithm is to obtain a 'sign matrix' for a set of univariate polynomials $p_1(x), \ldots, p_n(x)$. Such a matrix is a division of the real line into a (possibly empty) ordered sequence of m points $x_1 < x_2 < \cdots < x_m$ representing precisely the zeros of the polynomials, with the rows of the matrix representing, in alternating fashion, the points themselves and the intervals between adjacent pairs and the two intervals at the ends:

$$(-\infty, x_1), x_1, (x_1, x_2), x_2, \ldots, x_{m-1}, (x_{m-1}, x_m), x_m, (x_m, +\infty)$$

and columns representing the polynomials $p_1(x), \ldots, p_n(x)$, with the matrix entries giving the signs, either positive ($+$), negative ($-$) or zero (0), of each polynomial p_i at the points and on the intervals. For example, for the collection of polynomials:

$$\begin{aligned} p_1(x) &= x^2 - 3x + 2 \\ p_2(x) &= 2x - 3 \end{aligned}$$

the sign matrix looks like this:

Point/Interval	p_1	p_2
$(-\infty, x_1)$	$+$	$-$
x_1	0	$-$
(x_1, x_2)	$-$	$-$
x_2	$-$	0
(x_2, x_3)	$-$	$+$
x_3	0	$+$
$(x_3, +\infty)$	$+$	$+$

Note that x_1 and x_3 represent the roots 1 and 2 of $p_1(x)$ while x_2 represents 1.5, the root of $p_2(x)$. However the sign matrix contains no numerical information about the location of the points x_i, merely specifying the order of the roots of the various polynomials and what signs they take there and on the intervening intervals. It is easy to see that the sign matrix for a set of univariate polynomials $p_1(x), \ldots, p_n(x)$ is sufficient to answer any question of the form $\exists x.\ P[x]$ where the body $P[x]$ is quantifier-free and all atoms are of the form $p_i(x) \bowtie_i 0$ for any of the relations $=$, $<$, $>$, $\leq$, $\geq$ or their negations. We simply need to check each row of the matrix (point or interval) and see if one of them makes each atomic subformula true or false; the formula as a whole can then simply be "evaluated" by recursion.

In order to perform general quantifier elimination, we simply apply this basic operation to all the innermost quantified subformulas first (we can consider a universally quantified formula $\forall x.\ P[x]$ as $\neg(\exists x.\ \neg P[x])$ and eliminate from $\exists x.\ \neg P[x]$). This can then be iterated until all quantifiers are eliminated. The only difficulty is that the coefficients of a polynomial may now contain other variables as parameters; we will consider

the univariate case first for simplicity, and then consider the fairly straightforward generalization to the parametrized case.

We will explain the key parts of the algorithm both in English and with reference to the OCaml code. We use a simple representation of the sign matrix as a list of lists of sign values, the sign values belonging to a four-member enumerated type {`Positive`, `Negative`, `Zero`, `Nonzero`}. The top-level list corresponds to the sequence of points and intervals, and each sublist gives the sign values for the various polynomials there. For example, the sign matrix given above would be represented by

```
[[Positive; Negative];
 [Zero; Negative];
 [Negative; Negative];
 [Negative; Zero];
 [Negative; Positive];
 [Zero; Positive];
 [Positive; Positive]]
```

3.2 Computing the sign matrix

The following simple observation is key. To find the sign matrix for

$$p, p_1, \ldots, p_n$$

it suffices to find one for the set of polynomials

$$p', p_1, \ldots, p_n, q_0, q_1, \ldots, q_n$$

where p', which we will sometimes write p_0 for regularity's sake, is the derivative of p, and q_i is the remainder on dividing p by p_i. For suppose we have a sign matrix for the second set of polynomials. We can proceed as follows.

First, we split the sign matrix into two equally-sized parts, one for the $p', p_1, \ldots, p_n$ and one for the $q_0, q_1, \ldots, q_n$, but for now keeping all the points in each matrix, even if the corresponding set of polynomials has no zeros there. We can now infer the sign of $p(x_i)$ for each point x_i that is a zero of one of the polynomials $p', p_1, \ldots, p_n$, as follows. Since q_k is the remainder of p after division by p_k, $p(x) = s_k(x)p_k(x) + q_k(x)$ for some $s_k(x)$. Therefore, since $p_k(x_i) = 0$ we have $p(x_i) = q_k(x_i)$ and so we can derive the sign of p at x_i from that of the corresponding q_k. If the point x_i is not a zero of one of the $p', p_1, \ldots, p_n$, or we are dealing with an interval, we just arbitrarily assign `Nonzero`; it will be dealt with in the next step. The following code, given sign matrices `pd` for $p', p_1, \ldots, p_n$ and `qd` for $q_0, \ldots, q_n$, gives a corresponding sign matrix for $p, p', p_1, \ldots, p_n$, with the correct signs for p at the points, but not in general at intervals. (Here `index` gets the position index of the first occurrence of an element in a list, and `el` gets an indexed element.)

```
let inferpsign pd qd =
  try let i = index Zero pd in el i qd :: pd
  with Failure _ -> Nonzero :: pd;;
```

Now we can throw away the second sign matrix, giving signs for the $q_0, \ldots, q_n$, and retain the (partial) matrix for $p, p', p_1, \ldots, p_n$. We next 'condense' this matrix to remove points that are not zeros of one of the $p', p_1, \ldots, p_n$, but only of one of the q_i.

The signs of the $p', p_1, \ldots, p_n$ in an interval from which some other points have been removed can be read off from any of the subintervals in the original subdivision — they cannot change because there are no zeros for the relevant polynomials there.

```
let rec condense ps =
  match ps with
    int::pt::other -> let rest = condense other in
                      if mem Zero pt then int::pt::rest else rest
  | _ -> ps;;
```

Now we have a sign matrix with correct signs at all the points that are zeros of the the set of polynomials it involves, but with undetermined signs for p on the intervals, and the possibility that there may be additional zeros of p inside these intervals. But note that since there are certainly no zeros of p' inside the intervals, there can be at most one additional root of p in each interval. Whether there is one can be inferred, for an internal interval (x_i, x_{i+1}), by seeing whether the signs of $p(x_i)$ and $p(x_{i+1})$, determined in the previous step, are both nonzero and are different. If not, we can take the sign on the interval from whichever sign of $p(x_i)$ and $p(x_{i+1})$ is nonzero (we cannot have them both zero, since then there would have to be a zero of p' in between). Otherwise we insert a new point y between x_i and x_{i+1} which is a zero (only) of p, and infer the signs on the new subintervals (x_i, y) and (y, x_{i+1}) from the signs at the endpoints. Other polynomials have the same signs on (x_i, y), y and (y, x_{i+1}) that had been inferred for the original interval (x_i, x_{i+1}). For external intervals, we can use the same reasoning if we temporarily introduce new points $-\infty$ and $+\infty$ and infer the sign of $p(-\infty)$ by flipping the sign of p' on the lowest interval $(-\infty, x_1)$ and the sign of $p(+\infty)$ by copying the sign of p' on the highest interval $(x_n, +\infty)$. (Because the extremal behavior of polynomials is determined by the leading term, and those of p and p' are related by a positive multiple of x.) The following function assumes that these 'infinities' have been added first:

```
let rec inferisign ps =
  match ps with
    pt1::int::pt2::other ->
      let res = inferisign(pt2::other)
      and tint = tl int and s1 = hd pt1 and s2 = hd pt2 in
      if s1 = Positive & s2 = Negative then
        pt1::(Positive::tint)::(Zero::tint)::(Negative::tint)::res
      else if s1 = Negative & s2 = Positive then
        pt1::(Negative::tint)::(Zero::tint)::(Positive::tint)::res
      else if (s1 = Positive or s2 = Negative) & s1 = s2 then
        pt1::(s1::tint)::res
      else if s1 = Zero & s2 = Zero then
        failwith "inferisign: inconsistent"
      else if s1 = Zero then
        pt1::(s2 :: tint)::res
      else if s2 = Zero then
        pt1::(s1 :: tint)::res
      else failwith "inferisign: can't infer sign on interval"
  | _ -> ps;;
```

The overall operation is built up following the above lines. We structure it in such a way that it modifies a matrix and rather than returning it, passes it to a continuation function `cont`. As we will see later, this makes the overall implementation of the algorithm smoother. (Here `unzip` separates a list of pairs into two separate lists, `chop_list`

splits a list in two at a numbered position, `replicate k a` makes a list containing `k` copies of `a`, `tl` is the tail of a list and `butlast` returns all but the very last element.)

```
let dedmatrix cont mat =
  let n = length (hd mat) / 2 in
  let mat1,mat2 = unzip (map (chop_list n) mat) in
  let mat3 = map2 inferpsign mat1 mat2 in
  let mat4 = condense mat3 in
  let k = length(hd mat4) in
  let mats = (replicate k (swap true (el 1 (hd mat3))))::mat4@
             [replicate k (el 1 (last mat3))] in
  let mat5 = butlast(tl(inferisign mats)) in
  let mat6 = map (fun l -> hd l :: tl(tl l)) mat5 in
  cont(condense mat6);;
```

3.3 Multivariate polynomials

Note that this reasoning relies only on fairly straightforward observations of real analysis. Essentially the same procedure can be used even for multivariate polynomials, treating other variables as parameters while eliminating one variable. The only slight complication is that instead of literally dividing one polynomial s by another one p:

$$s(x) = p(x)q(x) + r(x)$$

we may instead have only a pseudo-division

$$a^k s(x) = p(x)q(x) + r(x)$$

where a is the leading coefficient of p, in general a polynomial in the other variables. In this case, to infer the sign of $p(x)$ from that of $r(x)$ where $q(x) = 0$ we need to know that $a \neq 0$ and what its sign is. Determining this may require a number of case-splits over signs or zero-ness of polynomials in other variables, complicating the formula if we then eliminate other variables. We will maintain an environment of sign hypotheses `sgns`, and when we perform pseudo-division, we will check that $a \neq 0$ and make sure that the signs of $s(x)$ and $r(x)$ are the same, by negating $r(x)$ or multiplying it by a when necessary, depending on how much we know about the sign of a and whether k is odd or even. (Here `pdivide` is a raw syntactic pseudo-division operation with no check on the nature of the divisor's head coefficient.)

```
let pdivides vars sgns q p =
  let s = findsign vars sgns (head vars p) in
  if s = Zero then failwith "pdivides: head coefficient is zero" else
  let (k,r) = pdivide vars q p in
  if s = Negative & k mod 2 = 1 then poly_neg r
  else if s = Positive or k mod 2 = 0 then r
  else poly_mul (tl vars) (head vars p) r;;
```

We will also need to case-split over positive/negative status of coefficients, and the following function fits a case-split into the continuation-passing framework; there is a similar function `split_zero` used as well:

```
let split_sign vars sgns pol cont_p cont_n =
  let s = findsign vars sgns pol in
  if s = Positive then cont_p sgns
  else if s = Negative then cont_n sgns
```

```
  else if s = Zero then failwith "split_sign: zero polynomial" else
  let ineq = Atom(R(">",[pol; Fn("0",[])])) in
  Or(And(ineq,cont_p (assertsign vars sgns (pol,Positive))),
     And(Not ineq,cont_n (assertsign vars sgns (pol,Negative))));;
```

We have explained a recursive algorithm for determining a sign matrix, but we haven't explained where to stop. If we reach a constant polynomial, then the sign will be determined (perhaps after case splitting) independent of the main variable, so we want to be able to insert a fixed sign at a certain place in a sign matrix. Again, we use a continuation-based interface:

```
let matinsert i s cont mat = cont (map (insertat i s) mat);;
```

The main loop will use the continuation to convert the finally determined sign matrix (of which there may be many variants because of case splitting) to a formula. However, note that because of the rather naive case-splitting, we may reach situations where an inconsistent set of sign assumptions is made — for example $a < 0$ and $a^3 > 0$ or just $a^2 < 0$. This can in fact lead to the 'impossible' situation that the sign matrix has two zeros of some $p(x)$ with no zero of $p'(x)$ in between them — which in `inferisign` will generate an exception. We do not want to actually fail here, but we are at liberty to return whatever formula we like, such as $\bot$. This is dealt with by the following exception-trapping function:

```
let trapout cont m =
  try cont m with Failure "inferisign: inconsistent" -> False;;
```

The main loop is organized as mutually recursive functions. The main function `matrix` assumes that the signs of all the leading coefficients of the polynomials are known, i.e. in `sgns`. If the set of polynomials is empty, we just apply the continuation to the trivial sign matrix, remembering the error trap. If there is a constant among the polynomials, we remove it and set up the continuation so the appropriate sign is re-inserted. Otherwise, we pick the polynomial with the highest degree, which will be the p in our explanation above, and recurse to `splitzero`, adding logic to rearrange the polynomials so that we can assume p is at the head of the list.

```
let rec matrix vars pols cont sgns =
  if pols = [] then trapout cont [[]] else
  if exists (is_constant vars) pols then
    let p = find (is_constant vars) pols in
    let i = index p pols in
    let pols1,pols2 = chop_list i pols in
    let pols' = pols1 @ tl pols2 in
    matrix vars pols' (matinsert i (findsign vars sgns p) cont) sgns
  else
    let d = itlist (max ** degree vars) pols (-1) in
    let p = find (fun p -> degree vars p = d) pols in
    let p' = poly_diff vars p and i = index p pols in
    let qs = let p1,p2 = chop_list i pols in p'::p1 @ tl p2 in
    let gs = map (pdivides vars sgns p) qs in
    let cont' m = cont(map (fun l -> insertat i (hd l) (tl l)) m) in
    splitzero vars qs gs (dedmatrix cont') sgns
```

The function `splitzero` simply case-splits over the zero status of the coefficients of the polynomials in the list `pols`, assuming those in `dun` are already fixed:

```
and splitzero vars dun pols cont sgns =
  match pols with
    [] -> splitsigns vars [] dun cont sgns
  | p::ops -> if p = Fn("0",[]) then
                let cont' = matinsert (length dun) Zero cont in
                splitzero vars dun ops cont' sgns
              else split_zero (tl vars) sgns (head vars p)
                   (splitzero vars dun (behead vars p :: ops) cont)
                   (splitzero vars (dun@[p]) ops cont)
```

When all polynomials are dealt with, we recurse to another level of splitting where we split over positive-negative status of the coefficients that are already determined to be nonzero:

```
and splitsigns vars dun pols cont sgns =
  match pols with
    [] -> dun
  | p::ops -> let cont' = splitsigns vars (dun@[p]) ops cont in
              split_sign (tl vars) sgns (head vars p) cont' cont'
```

That is the main loop of the algorithm; we start with a continuation that will perform an appropriate test on the sign matrix entries for each literal in the formula, and we construct the sign matrix for all polynomials that occur in the original formula.

4 Proof-producing implementation

The proof-producing version, by design, follows the same structure. In this section we concentrate on interesting design decisions for this variant.

4.1 Polynomials

Our canonical representation of polynomials is as lists of coefficients with the constant term first. For example, the polynomial x^3-2x+6 is represented by the list $[6;-2;0;1]$. The corresponding 'evaluation' function is simply expressed as a primitive recursive definition over lists:

```
⊢ (poly [] x = &0) ∧
  (poly (CONS h t) x = h + x * poly t x)
```

This representation is used in a nested fashion to encode multivariate polynomials. A key point is that we can prove many analytical theorems such as special intermediate-value properties generically for all polynomials just by using `poly l` for a general list of reals `l` (the proofs usually proceed by induction over lists). Thus we can avoid proving many special cases for the actual polynomials we use: they are deduced by a single primitive inference step of variable instantiation from the generic versions.

4.2 Data Structures

The first difficult choice we encountered was how to represent our current knowledge about the state of the sign matrix. We were faced with the task of organizing the following information, for example, giving a partial sign matrix for the polynomials

$$p_0(x) = x^2$$
$$p_1(x) = x - 1$$

(Here, x_0 and x_1 correspond to the roots 0 of $\lambda x.\ x^2$ and 1 of $\lambda x.\ x - 1$ respectively.)

$$\forall x.\ x < x_0 \Rightarrow p_0(x) > 0$$
$$\forall x.\ x < x_0 \Rightarrow p_1(x) < 0$$
$$p_0(x_0) = 0$$
$$p_1(x_0) < 0$$
$$\forall x.\ x_0 < x < x_1 \Rightarrow p_0(x) > 0$$
$$\forall x.\ x_0 < x < x_1 \Rightarrow p_1(x) < 0$$
$$p_0(x_1) > 0$$
$$p_1(x_1) = 0$$
$$\forall x.\ x_1 < x \Rightarrow p_0(x) > 0$$
$$\forall x.\ x_1 < x \Rightarrow p_1(x) > 0$$

As the matrices become larger (which is immediate due to the exponential nature of the procedure), the task of managing these facts using simple data types like lists and products becomes daunting. Instead, we chose to define a series of predicates that allow us to organize this data in a succinct fashion. We begin by defining an enumerated type of signs with members `Zero`, `Pos`, `Neg`, `Nonzero`, `Unknown`. The additional element `Unknown` is useful in order to be able to make rigorous proven statements at intermediate steps, whereas in the original we could just say 'the setting of this sign may be wrong now but we'll fix it in the next step'. We then define a predicate which, given a domain and a polynomial, interprets the sign:

$$\texttt{interpsign}\ S\ p\ \texttt{Zero} := (\forall x.x \in S \Rightarrow (p(x) = 0))$$
$$\texttt{interpsign}\ S\ p\ \texttt{Pos} := (\forall x.x \in S \Rightarrow (p(x) > 0))$$
$$\texttt{interpsign}\ S\ p\ \texttt{Neg} := (\forall x.x \in S \Rightarrow (p(x) < 0))$$
$$\texttt{interpsign}\ S\ p\ \texttt{Nonzero} := (\forall x.x \in S \Rightarrow (p(x) \neq 0))$$
$$\texttt{interpsign}\ S\ p\ \texttt{Unknown} := (\forall x.x \in S \Rightarrow (p(x) = p(x)))$$

Now, the previous set of formulas can be written as follows:

$$\texttt{interpsign}\ (\lambda x.x < x_0)\ p_0\ \texttt{Pos}$$
$$\texttt{interpsign}\ (\lambda x.x < x_0)\ p_1\ \texttt{Neg}$$
$$\texttt{interpsign}\ (\lambda x.x = x_0)\ p_0\ \texttt{Zero}$$
$$\texttt{interpsign}\ (\lambda x.x = x_0)\ p_1\ \texttt{Neg}$$
$$\texttt{interpsign}\ (\lambda x.x_0 < x < x_1)\ p_0\ \texttt{Pos}$$
$$\texttt{interpsign}\ (\lambda x.x_0 < x < x_1)\ p_1\ \texttt{Neg}$$
$$\texttt{interpsign}\ (\lambda x.x = x_1)\ p_0\ \texttt{Pos}$$
$$\texttt{interpsign}\ (\lambda x.x = x_1)\ p_1\ \texttt{Zero}$$
$$\texttt{interpsign}\ (\lambda x.x_1 < x)\ p_0\ \texttt{Pos}$$
$$\texttt{interpsign}\ (\lambda x.x_1 < x)\ p_1\ \texttt{Pos}$$

But these formulas are of a very predictable form, so we can use HOL Light 's cababilities for making primitive recursive definitions:

$$\texttt{ALL2}\ P\ []\ l2 \Leftrightarrow (l2 = [])$$
$$\texttt{ALL2}\ P\ (\texttt{CONS}\ h1\ t1)\ l2 \Leftrightarrow$$
$$\texttt{if}\ l2 = []\ \texttt{then}\ F\ \texttt{else}\ (P\ h1\ (\texttt{HD}\ l2)) \wedge (\texttt{ALL2}\ P\ t1\ (\texttt{TL}\ l2))$$

This allows us to compact the formulas for a given set with another predicate:

$$\texttt{interpsigns}\ polys\ S\ signs = \texttt{ALL2}\ (\texttt{interpsign}\ S)\ polys\ signs$$

Our formulas can now be represented slightly more succinctly:

$$\texttt{interpsigns}\ (\lambda x.x < x_0)\ [p_0, p_1]\ [\texttt{Pos}, \texttt{Neg}]$$
$$\texttt{interpsigns}\ (\lambda x.x = x_0)\ [p_0, p_1]\ [\texttt{Zero}, \texttt{Neg}]$$
$$\texttt{interpsigns}\ (\lambda x.x_0 < x < x_1)\ [p_0, p_1]\ [\texttt{Pos}, \texttt{Neg}]$$
$$\texttt{interpsigns}\ (\lambda x.x = x_1)\ [p_0, p_1]\ [\texttt{Pos}, \texttt{Zero}]$$
$$\texttt{interpsign}\ (\lambda x.x_1 < x)\ [p_0, p_1]\ [\texttt{Pos}, \texttt{Pos}]$$

Now, given a predicate `OrderedList` which indicates that the points in the list are sorted, and a function `PartitionLine` that breaks the real line into intervals based on the points of the list, we can represent the entire matrix with a final predicate.

$$\texttt{interpmat}\ points\ polys\ signs =$$
$$\texttt{OrderedList}\ points \wedge$$
$$\texttt{ALL2}\ (\texttt{interpsigns}\ polys)\ (\texttt{PartitionLine}\ points)\ signs$$

Thus, our entire set of formulas is represented by the simple formula

$$\texttt{interpmat}\ [x_0, x_1]\ [p_0, p_1]\ [[\texttt{Pos}, \texttt{Neg}], [\texttt{Zero}, \texttt{Neg}], [\texttt{Pos}, \texttt{Neg}], [\texttt{Pos}, \texttt{Zero}], [\texttt{Pos}, \texttt{Pos}]]$$

As the sign matrix is the primary data structure in the algorithm, this succinct representation makes the implementation much smoother than dealing with the formulas individually.

One potential drawback to this approach is the time spent assembling and disassembling the representation to extract or add formulas. While in a high level programming language this would not be a concern, for large matrices this is a substantial amount of term rewriting.

On the other hand, the representation can also speed up rewriting. For example, suppose we've deduced that $p_2 = p_0$ and would like to replace p_0 by p_2 in the sign matrix. This is a trivial one step rewrite for our representation, while many rewrites would be required to achieve the same result if the matrix were represented by individual formulas.

4.3 Existential Formulas

Creating the sign matrix involves instantiating a large number of existential theorems, such as a theorem used for adding the "points at infinity":

$$(\forall x.\ x > y \Rightarrow p'(x) < 0) \Rightarrow \exists Y. Y > y \wedge (\forall y.\ y \geq Y \Rightarrow p(y) < 0)$$

These existential formulas are instantiated by the usual method of choosing an unused variable name for the instance and assuming the body of the theorem. The assumptions are then eliminated using the usual existential elimination rule of natural deduction. This is unproblematic in the univariate case, where there are no splits in the control flow due to sign variations. In the general case, though, the existential theorems themselves can depend on sign assumptions. Thus we must be careful to carry these existential theorems through every step of the algorithm until we reach an appropriate place to eliminate them. This was a common characteristic of adapting the procedure given above to produce proofs. A number of environments and continuations were necessary to pass necessary theorems both up and down the tree of sign splits.

4.4 Effort Comparison

Generating proofs for each step of the code above was a significant challenge. To compare the procedures, we first consider the amount of code that was necessary to implement the procedure. The code given above, when augmented with the necessary library functions, runs around 600 lines. In contrast, our version runs around 4000. This does not include the additional 4000 lines of proof scripts, proving lemmas needed at various points in the procedure. An illustrative (and perhaps amusing) example in the algorithm presented above occurs in `dedmatrix` where the "points at infinity" are added and removed smoothly in 3 lines of code. Our subroutine to add these points is 90 lines, based on lemmas requiring 3000 lines of proof guaranteeing their existence. This is an extreme, but not uncharacteristic example of the difference in effort required.

5 Results

It is well-known that quantifier elimination for the reals is in general computationally intractable, both in theoretical complexity [31] and in the limited success on real applications. So we cannot start with high expectations of routinely solving really interesting problems. This applies with all the more force since proof production makes the algorithm considerably slower, apparently about 3 orders of magnitude for our current prototype.

However, in the field of interactive theorem proving, the algorithm could be an important tool. A great deal of time can be spent proving trivial facts of real arithmetic that do not fall into one of the well known decidable (and implemented) subsets, such as linear arithmetic, or linear programming. One author spent many hours proving simple lemmas in preparation for implementing this procedure. Some indicative examples are

$$\forall x \forall y.\ x \cdot y > 0 \Leftrightarrow (x > 0 \wedge y > 0) \vee (x < 0 \wedge y < 0)$$
$$\forall x \forall y.\ x \cdot y < 0 \Leftrightarrow (x > 0 \wedge y < 0) \vee (x < 0 \wedge y > 0)$$
$$\forall x \forall y.\ x < y \Rightarrow \exists z.\ x < z \wedge z < y$$

Our procedure easily dispenses with such problems, and compares favorably with the time it takes to prove such problems "by hand". Thus, it could be a potentially valuable tool in the day to day use of theorem provers like HOL Light.

5.1 Times

Without further ado, we give some of the problems the algorithm can solve and running times. The procedure is written in OCaml and was run uncompiled on a 3GHz Pentium 4 processor running Linux kernel 2.4. Times are in seconds unless otherwise indicated.

Univariate Examples We've arranged the results into loose categories. We first consider some routine univariate examples, the results collected in Table 1. A final pair of examples demonstrates the key bound properties of the first two "Chebyshev Polynomials" which are useful in function approximation; these had running times of 24 and 65 seconds respectively.

$$\forall x.\ -1 \leq x \wedge x \leq 1 \Rightarrow -1 \leq 2x^2 - 1 \wedge 2x^2 - 1 \leq 1$$
$$\forall x.\ -1 \leq x \wedge x \leq 1 \Rightarrow -1 \leq 4x^3 - 3x \wedge 4x^3 - 3x \leq 1$$

Multivariate Examples

Category	Formula	Result	Running Time
Linear	$\exists x.\ x - 1 > 0$	T	1.5
Linear	$\exists x.\ 3 - x > 0 \wedge x - 1 > 0$	T	4.0
Quadratic	$\exists x.\ x^2 = 0$	T	3.0
Quadratic	$\exists x.\ x^2 + 1 = 0$	F	2.9
Quadratic	$\exists x.\ x^2 - 2x + 1 = 0$	T	3.1
Cubic	$\exists x.\ x^3 - 1 = 0$	T	4.9
Cubic	$\exists x.\ x^3 - 3x^2 + 3x - 1 > 0$	T	5.1
Cubic	$\exists x.\ x^3 - 4x^2 + 5x - 2 > 0$	T	10.6
Cubic	$\exists x.\ x^3 - 6x^2 + 11x - 6 = 0$	T	11.2
Quartic	$\exists x. x^4 - 1 > 0$	T	7.2
Quartic	$\exists x. x^4 + 1 < 0$	F	6.1
Quartic	$\exists x. x^4 - x^3 = 0$	T	27.6
Quartic	$\exists x. x^4 - 2 * x^2 + 2 = 0$	T	32.6
Quintic	$\exists x. x^5 - 15 * x^4 + 85 * x^3 - 225 * x^2 + 274 * x - 120 = 0$	T	600

Table 1. Runtimes on simple univariate examples

– Here is an instance of a more complicated quantifier structure. Our implementation returns the theorem in 63 seconds.

$$(\forall a\ f\ k.\ (\forall e.\ k < e \Rightarrow f < a \cdot e) \Rightarrow f \leq a \cdot k)$$

– Here is an example arising as a polynomial termination ordering for a rewrite system for group theory, which takes around 2 minutes.

$$1 < 2 \wedge (\forall x.\ 1 < x \Rightarrow 1 < x^2) \wedge \\ (\forall x\ y.\ 1 < x \wedge 1 < y \Rightarrow 1 < x(1 + 2y))$$

– We can use open formulas to determine when polynomials have roots, as in the case mentioned above of a quadratic polynomial, $\exists x.\ ax^2 + bx + c = 0$ The following identity is established in 27 seconds.

```
val it : thm =
  |- (?x. a * x pow 2 + b * x + c = &0) <=>
     (&0 + a * &1 = &0) /\
     ((&0 + b * &1 = &0) /\ (&0 + c * &1 = &0) \/
      ~(&0 + b * &1 = &0) /\ (&0 + b * &1 > &0 \/ &0 + b * &1 < &0)) \/
     ~(&0 + a * &1 = &0) /\
     (&0 + a * &1 > &0 /\
      ((&0 + a * ((&0 + b * (&0 + b * -- &1)) + a * (&0 + c * &4)) = &0) \/
       ~(&0 + a * ((&0 + b * (&0 + b * -- &1)) + a * (&0 + c * &4)) = &0) /\
       &0 + a * ((&0 + b * (&0 + b * -- &1)) + a * (&0 + c * &4)) < &0) \/
      &0 + a * &1 < &0 /\
      ((&0 + a * ((&0 + b * (&0 + b * -- &1)) + a * (&0 + c * &4)) = &0) \/
       ~(&0 + a * ((&0 + b * (&0 + b * -- &1)) + a * (&0 + c * &4)) = &0) /\
       &0 + a * ((&0 + b * (&0 + b * -- &1)) + a * (&0 + c * &4)) > &0))
```

While this is not particularly readable, it does give the necessary and sufficient conditions. We can express the answer in a somewhat nicer form by "guessing", and the proof takes 9940 seconds:

$$\begin{array}{l}\forall a \forall b \forall c.(\exists x.ax^2 + bx + c = 0) \Leftrightarrow \\ \quad (((a = 0) \wedge ((b \neq 0) \vee (c = 0))) \vee \\ \quad (a \neq 0) \wedge b^2 \geq 4ac)\end{array}$$

– Robert Solovay has shown us a method by which formulas over general real vector spaces can be reduced to the present subset of reals. Consider the following formula, where x and y are vectors and u a real number, $x \cdot y$ is the inner (dot) product and $||x||$ is the norm (length) of x:

$$\forall x\ y.\ x \cdot y > 0 \Rightarrow \exists u.\ 0 < u \wedge ||uy - x|| < ||x||$$

Our implementation of Solovay's procedure returns the following formula over the reals that provably implies the original. (Note that the body can be subjected to some significant algebraic simplification, but this gets handled anyway by our transition to canonical polynomial form.) Our procedure proves this in 200 seconds.

$$\begin{array}{l}\forall a\ b\ c.\ 0 \leq b \wedge 0 \leq c \wedge 0 < ac \\ \quad \Rightarrow \exists u.\ 0 < u \wedge u(uc - ac) - (uac - (a^2c + b)) < a^2c + b\end{array}$$

6 Future Work

The underlying algorithm is quite naive, and could be improved in many ways at relatively little cost in complexity. One very promising improvement is to directly exploit equations to substitute. At its simplest, if we are eliminating an existential quantifier from a conjunction containing an equation with the variable on one side, we can simply replace the variable with the other side of the equation. (At present, our algorithm uses the inefficient general sign-matrix process even when such obvious simplifications could be made.) Slightly more complicated methods can yield very good results for low-degree equations like quadratics [32]. More generally, even more complicated higher-degree equations can be used to substitute, and we can even try to factor. For example, consider the assertion that the logistic map $x \mapsto rx(1 - x)$ has a cycle with period 2:

$$\exists x.\ 0 \leq x \wedge x \leq 1 \wedge r(rx(1 - x))(1 - rx(1 - x)) = x \wedge \neg(rx(1 - x) = x)$$

By factoring the equation and then using the remaining factor to substitute, we can reach the following formula, where the degree of x has been reduced, making the problem dramatically easier for the core algorithm:

$$\exists x.\, 0 \leq x \wedge x \leq 1 \wedge r^2x^2 - r(1+r)x + (1+r) = 0 \wedge \neg(2rx = 1+r)$$

The translation to a proof-producing version was done quite directly, and there is probably considerable scope for improvement by making some of the inference steps more efficient. In the last example, the HOL Light implementation runs over 10^3 slower than the unchecked version. It seems that this gap can be significantly narrowed.

One interesting continuation of the current work would be to see how easily our implementation could be translated to another theorem prover such as Isabelle [25]. Finally, there is the potential to use this procedure in a fully automated combined decision procedure environment such as CVC-Lite [1]. We have not explored these lines of research in any detail.

7 Conclusion

It is difficult to foresee the practical benefits of using general decision procedures such as this one in the field of interactive theorem proving. As this case study shows, even when they exist, it is not at all clear whether, due to complexity constraints, they will be applicable to even moderately difficult problems. Considering the examples given above, one might even dismiss such procedures outright as entirely too inefficient. For the *user* of such a system, however, it is procedures like this one which automate tedious low level tasks that make the process of theorem proving useful and enjoyable, or at least tolerable.

In conclusion, the work described above can be seen in two different lights. On the one hand, it is a rather inefficient implementation of an algorithm which, while mathematically and philosophically interesting, and theoretically applicable to an enormous range of difficult problems, is not yet practically useful for those problems. On the other hand, it can be viewed as another tool in the (human) theorem prover's tool chest. One that, given the wide range of applications of the real numbers in theorem proving, could be an important *practical* achievement.

Acknowledgments

The first author would like to thank Frank Pfenning and for the Logosphere grant that helped to support this work. The second author is grateful to Loïc Pottier for first telling him about the Hörmander procedure.

References

1. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, Boston, MA, 2004. Springer-Verlag.

2. M. Blum. Program result checking: A new approach to making programs more reliable. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Automata, Languages and Programming, 20th International Colloquium, ICALP93, Proceedings*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 1993. Springer-Verlag.
3. J. Bochnak, M. Coste, and M.-F. Roy. *Real Algebraic Geometry*, volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1998.
4. R. J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1993. Author's PhD thesis.
5. B. F. Caviness and J. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and monographs in symbolic computation. Springer-Verlag, 1998.
6. P. J. Cohen. Decision procedures for real and p-adic fields. *Communications in Pure and Applied Mathematics*, 22:131–151, 1969.
7. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Second GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183, Kaiserslautern, 1976. Springer-Verlag.
8. M. Davis. A computer program for Presburger's algorithm. In *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University*, pages 215–233. Institute for Defense Analyses, Princeton, NJ, 1957. Reprinted in [28], pp. 41–48.
9. E. Engeler. *Foundations of Mathematics: Questions of Analysis, Geometry and Algorithmics*. Springer-Verlag, 1993. Original German edition *Metamathematik der Elementarmathematik* in the *Series Hochschultext*.
10. L. Gårding. *Some Points of Analysis and Their History*, volume 11 of *University Lecture Series*. American Mathematical Society / Higher Education Press, 1997.
11. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
12. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
13. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.
14. J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
15. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
16. J. Harrison. Complex quantifier elimination in HOL. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, pages 159–174. Division of Informatics, University of Edinburgh, 2001. Published as Informatics Report Series EDI-INF-RR-0046. Available on the Web at http://www.informatics.ed.ac.uk/publications/report/0046.html.
17. L. Hörmander. *The Analysis of Linear Partial Differential Operators II*, volume 257 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1983.
18. W. A. Hunt, R. B. Krug, and J. Moore. Linear and nonlinear arithmetic in ACL2. In D. Geist, editor, *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag.
19. G. Kreisel and J.-L. Krivine. *Elements of mathematical logic: model theory*. Studies in Logic and the Foundations of Mathematics. North-Holland, revised second edition, 1971.

First edition 1967. Translation of the French 'Eléments de logique mathématique, théorie des modeles' published by Dunod, Paris in 1964.
20. R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, pages 170–176, University of California at Davis, Davis CA, USA, 1991. IEEE Computer Society Press.
21. A. Mahboubi and L. Pottier. Elimination des quantificateurs sur les réels en Coq. In Journées Francophones des Langages Applicatifs (JFLA), available on the Web from http://pauillac.inria.fr/jfla/2002/actes/index.html08-mahboubi.ps, 2002.
22. C. Michaux and A. Ozturk. Quantifier elimination following Muchnik. Université de Mons-Hainaut, Institute de Mathématique, Preprint 10, http://w3.umh.ac.be/math/preprints/src/Ozturk020411.pdf, 2002.
23. G. C. Necula. Proof-carrying code. In *Conference record of POPL'97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
24. P. Parrilo. Semidefinite programming relaxations for semialgebraic problems. Available from the Web at citeseer.nj.nec.com/parrilo01semidefinite.html, 2001.
25. L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.
26. H. Schoutens. Muchnik's proof of Tarski-Seidenberg. Notes available from http://www.math.ohio-state.edu/ schoutens/PDF/Muchnik.pdf, 2001.
27. A. Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 60:365–374, 1954.
28. J. Siekmann and G. Wrightson, editors. *Automation of Reasoning — Classical Papers on Computational Logic, Vol. I (1957-1966)*. Springer-Verlag, 1983.
29. C. Sturm. Mémoire sue la résolution des équations numériques. *Mémoire des Savants Etrangers*, 6:271–318, 1835.
30. A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948; prepared for publication by J. C. C. McKinsey. Reprinted in [5], pp. 24–84.
31. N. N. Vorobjov. Deciding consistency of systems of polynomial in exponent inequalities in subexponential time. In T. Mora and C. Traverso, editors, *Proceedings of the MEGA-90 Symposium on Effective Methods in Algebraic Geometry*, volume 94 of *Progress in Mathematics*, pages 491–500, Castiglioncello, Livorno, Italy, 1990. Birkhäuser.
32. V. Weispfenning. Quantifier elimination for real algebra — the quadratic case and beyond. *Applicable Algebra in Engineering Communications and Computing*, 8:85–101, 1997.