Real quantifier elimination

John Harrison

Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro OR 97124
johnh@ichips.intel.com

Abstract. This material is taken from the author's current manuscript version of "Introduction to Logic and Automated Theorem Proving".

We now consider a similar theory of *real* arithmetic with addition and multiplication. A decision procedure for this theory, based on quantifier elimination, was first demonstrated by Tarski (1951).[1] However, Tarski's procedure, a generalization of the classical technique due to Sturm (1835) for finding the number of real roots of a univariate polynomial, was both difficult to understand and highly inefficient in practice. Seidenberg (1954) gave a simpler algorithm; indeed the possibility of quantifier elimination for this theory is often dually attributed as 'Tarski-Seidenberg'. Other relatively simple algorithms were given by Cohen (1969) and by Kreisel and Krivine (1971). 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introduced by Collins (1976).[2] The rather simple algorithm we develop here is from Hörmander (1983) based on an unpublished manuscript by Paul Cohen.

In our language we will allow both equations $s = t$ and inequalities $s < t$, $s \leq t$, $s > t$ and $s \geq t$. Our algorithm necessarily has a somewhat different flavour from the complex number procedure, not just because of the presence of inequalities, but because the reals are not algebraically closed. For example, since the quadratic equation $x^2 + 1 = 0$ has no solution over $\mathbb{R}$, the following are both valid, yet there is no simple divisibility relation between powers of the antecedent and consequent polynomials:

$$\forall x.\ x^2 + 1 = 0 \Longrightarrow x + 2 = 0$$
$$\forall x.\ x^3 + 2x^2 + x + 2 = 0 \Longrightarrow x^2 + 4x + 4 = 0$$

[1] Tarski actually discovered the procedure in 1930, but it remained unpublished for many years afterwards. Tarski's procedure, and the one we will describe, work not only for the reals but for any 'real closed field', a point to which we will return.

[2] A related technique was earlier proposed by Łojasiewicz (1964). Another relatively efficient method was developed by L. Monk, working with Solovay. This is briefly described by Rabin (1991) and in more detail in Monk's UC Berkeley PhD thesis.

The algorithm will essentially use ordering properties, and we will freely exploit basic facts about polynomials over the reals.[3] Some of our reasoning will involve derivatives, so we start with a function to differentiate a polynomial with respect to the top variable. The derivative of a $p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$ is just $p'(x) = c_1 + 2c_2 + \cdots + nc_n x^{n-1}$, but we need to operate on the canonical form. This auxiliary function takes as additional parameters the top variable x (as a term) and the implicit power of x by which the polynomial is multiplied; this determines the multiplier for the first coefficient:

```
let rec poly_diffn x n p =
  match p with
    Fn("+",[c; Fn("*",[y; q])]) when y = x ->
        Fn("+",[poly_cmul(Int n) c; Fn("*",[x; poly_diffn x (n+1) q])])
  | _ -> poly_cmul(Int n) p;;
```

Now to differentiate a polynomial $p(x) = c + x \cdot q(x)$, we just apply the auxiliary function to $q(x)$ with $n = 1$; if $p(x)$ is constant we just return zero.

```
let poly_diff vars p =
  match p with
    Fn("+",[c; Fn("*",[Var x; q])]) when x = hd vars ->
         poly_diffn (Var x) 1 q
  | _ -> zero;;
```

The key component of the quantifier elimination algorithm is a procedure to obtain a 'sign matrix' for a set of univariate polynomials $p_1(x), \ldots, p_n(x)$. Such a matrix is based on 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:

$$p_1(x) = x^2 - 3x + 2$$
$$p_2(x) = 2x - 3$$

the sign matrix looks like this:

[3] Most of these are familiar from elementary calculus applied to differentiable functions of one variable. With more work, the properties we need can be deduced just from the real-closed field axioms.

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	$+$
$(x_3, +\infty)$	$+$	$+$

Here x_1 and x_3 represent the roots 1 and 2 of $p_1(x)$ while x_2 represents $3/2$, the root of $p_2(x)$. However the sign matrix contains no numerical information about the location of the points x_i, merely specifying their order and what signs the various polynomials take on each point and each intermediate interval. Crucially, 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. Each relation $\bowtie$ is associated with a set of signs for p for which $p \bowtie 0$ holds:

```
let rel_signs =
  ["=",[Zero]; "<=",[Zero;Negative]; ">=",[Zero;Positive];
   "<",[Negative]; ">",[Positive]];;
```

Now, given an association list `pmat` of polynomials with their signs, we can evaluate a formula by just:

```
let testform pmat fm =
  eval fm (fun (R(a,[p;z])) -> mem (assoc p pmat) (assoc a rel_signs));;
```

As we will see, the generalization to multivariate polynomials is straightforward, so being able to find the sign matrix is the core of our enterprise. And a fairly simple recursive algorithm to find sign matrices can be based on the following observation. We can construct the sign matrix for the polynomials:

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

given a sign matrix for the following polynomials, where p' is the derivative of p, and each q_i is the remainder on dividing p by p_i (where $p_0 = p'$ for regularity):

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

The procedure for deriving the sign matrix for the first set, given one for the second, is 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 the moment keeping all the points, even if no polynomial in one set has a zero at some of them. 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_k, as follows. Since q_k is the remainder on dividing p by p_k,

we have $p(x) = s_k(x)p_k(x) + q_k(x)$ 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 assign `Nonzero`; such points will be eliminated in the next step. The following code implements this process for two corresponding rows `pd` and `qd` of the sign matrices for $p', p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ respectively.

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

Having applied this to all rows, we throw away the second sign matrix, giving signs for the $q_0, \ldots, q_n$, and retain the (partial) matrix for $p, p', p_1, \ldots, p_n$, which we 'condense' to remove points that are not zeros of one of the $p', p_1, \ldots, p_n$. 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, but undetermined signs for p on the intervals, and the possibility that there may be additional zeros of p inside these intervals. However, note that there can be *at most one* zero of p in each interval, even including its endpoint(s). For if there were two zeros, then p would reach a maximum or minimum somewhere in between them, contradicting the fact that p' is nonzero on the interior of the interval.

Consider first an internal interval (x_i, x_{i+1}). By the observation above, if $p(x_i) = 0$ or $p(x_{i+1}) = 0$ we know that there can be no other zero in the interval. If both $p(x_i)$ and $p(x_{i+1})$ are nonzero and their signs are different then there *is* a zero of p in the interval, by the intermediate value property. Finally, if the signs are both nonzero but are the same, there is no root in the interval, because in that case p would reach a maximum or minimum there (whether it crosses or just touches the x-axis), and this is impossible since $p' \neq 0$. To summarize, there is one root of p inside the interval if the signs of $p(x_i)$ and $p(x_{i+1})$ are both nonzero and different, and there is no root otherwise.

What about the two semi-infinite intervals? For sufficiently large $|x|$, a polynomial is dominated by the term of highest degree, and if $p(x) \sim a_n x^n$ we have $p'(x) \sim n a_n x^{n-1}$, so the ratio between the two eventually has positive sign as $x \to +\infty$ and negative sign as $x \to -\infty$. Let us temporarily introduce pseudo-endpoints $-\infty$ and $+\infty$ to denote 'points at infinity'. Based on the above observation, we define 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 $(x_n, +\infty)$. Now exactly the same decision method works for this case too, which makes the implementation more regular.

The following function implements these observations to complete the partial sign matrix, assuming that the 'points at infinity' have been added first. When this is called, the first three elements of `ps` are the lists of polynomial signs for respectively the leftmost point, the interval following it, and the next point to its right. We pick out the signs of p (the head of each list) at the left (`l`) and right (`r`) endpoints of the interval. It should actually be impossible for both signs to be zero, since that would imply a point of zero derivative between. And we hope never to encounter just Nonzero; by design we will always have a more precise sign whenever `inferisign` function is used. Otherwise, if just one sign is zero, we infer the sign on the interval from the sign at the nonzero end. If both are negative or both positive, we infer the sign from `l` (we could equally well use `r`). The more complex case is where `l` and `r` are opposites, and we insert a new point and its surrounding intervals. The signs of p on the new subintervals are taken from the corresponding endpoints, and it is zero at the new point. Nothing changes for the other polynomials throughout the original interval, so we just duplicate `ints` for them. In each case we recursively call `inferisign` to deal with the remaining points and intervals. And finally, when there are fewer than three elements, we assume we have reached the rightmost endpoint, so there are no intervals to infer the sign of p on, and we return the original sign matrix unchanged.

```
let rec inferisign ps =
  match ps with
    ((l::ls) as x)::(_::ints)::((r::rs)::xs as pts) ->
       (match (l,r) with
          (Zero,Zero) -> failwith "inferisign: inconsistent"
        | (Nonzero,_)
        | (_,Nonzero) -> failwith "inferisign: indeterminate"
        | (Zero,_) -> x::(r::ints)::inferisign pts
        | (_,Zero) -> x::(l::ints)::inferisign pts
        | (Negative,Negative)
        | (Positive,Positive) -> x::(l::ints)::inferisign pts
        | _ -> x::(l::ints)::(Zero::ints)::(r::ints)::inferisign pts)
  | _ -> ps;;
```

Now we're ready for the overall function to convert a sign matrix `mat` for $p', p_1, \ldots, p_n, q_0, q_1, \ldots, q_n$ into one for $p, p_1, \ldots, p_n$. Rather than returning the result, it applies the given continuation function `cont` to it, since this fits in with the later code structure. Otherwise it's just a question of putting together the earlier pieces. We set $l = n + 1$, and apply `inferpsign` to all rows of the matrix, first splitting them into the pieces for $p', p_1, \ldots, p_n$ and for $q_0, q_1, \ldots, q_n$. After condensation to remove extraneous points, we get a partial sign matrix `mat1` for $p, p', p_1, \ldots, p_n$. The points at infinity are added, just for p since nothing else

will be looked at, to give `mat2`. We then infer the signs on the intervals and remove the points at infinity again to give `mat3`. Finally, we remove p' from this matrix, condense again to remove points that were just zeros of p', and apply the continuation to the result.

```
let dedmatrix cont mat =
  let l = length (hd mat) / 2 in
  let mat1 = condense(map (inferpsign ** chop_list l) mat) in
  let mat2 = [swap true (el 1 (hd mat1))]::mat1@[[el 1 (last mat1)]] in
  let mat3 = butlast(tl(inferisign mat2)) in
  cont(condense(map (fun l -> hd l :: tl(tl l)) mat3));;
```

The reasoning underlying `dedmatrix` is based 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 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. As with the complex numbers, we will need to perform case splits over polynomials in other variables to make sure $a \neq 0$. Even then, to infer the sign of r from that of s, we need to know the sign of a^k. Our solution is an enhanced pseudo-division function that ensures that r has the same sign as s as follows. We obtain the head coefficient a of $p(x)$ and perform pseudo-division as usual, say $a^k s(x) = p(x)q(x) + r(x)$. We then examine what we know from the context about the sign of a. If it is zero, we fail, and if the context does not determine it, `findsign` will fail. Otherwise if we know either that $a > 0$ or that k is even, we have $a^k > 0$ and can safely return $r(x)$. Otherwise, k must be odd. If we know $a < 0$, then also $a^k < 0$ so we need to return $-r(x)$. Otherwise, all we know is $a \neq 0$, so we implicitly multiply through again by a and return $ar(x)$; note that $a^{k+1}s(x) = ap(x)q(x) + ar(x)$, and since k is odd, $k + 1$ is even.

```
let pdivide_pos vars sgns s p =
  let a = head vars p and (k,r) = pdivide vars s p in
  let sgn = findsign sgns a in
  if sgn = Zero then failwith "pdivide_pos: zero head coefficient"
  else if sgn = Positive or k mod 2 = 0 then r
  else if sgn = Negative then poly_neg r else poly_mul vars a r;;
```

We will also need to case-split over positive/negative status of coefficients, and the following function is analogous to the function `split_zero` that we

wrote for the complex numbers and will shortly use again. It is assumed that by the time we use this function, we already know from the context at least that the polynomial concerned is nonzero.

```
let split_sign sgns pol cont =
  match findsign sgns pol with
    Nonzero -> let fm = Atom(R(">",[pol; zero])) in
               Or(And(fm,cont(assertsign sgns (pol,Positive))),
                  And(Not fm,cont(assertsign sgns (pol,Negative))))
  | _ -> cont sgns;;
```

In the later algorithm, the most convenient thing is to perform a three-way case-split over the zero, positive or negative cases, but call the same continuation on the positive and negative cases:

```
let split_trichotomy sgns pol cont_z cont_pn =
  split_zero sgns pol cont_z (fun s' -> split_sign s' pol cont_pn);;
```

Sign matrix determination is now implemented by a set of three mutually recursive functions. The first function `casesplit` takes two lists of polynomials: `dun` (so named because 'done' is a reserved word in OCaml) is the list whose head coefficients have known sign, and `pols` is the list that are to be checked. As soon as we have determined all the head coefficient signs, we call `matrix`. For each polynomial `p` in the list `pols` we perform appropriate case splits. In the zero case we chop off its head coefficient and recurse, and in the other cases we just add it to the 'done' list. But if any of the polynomials is a constant with respect to the top variable, we recurse to a `delconstant` function to remove it.

```
let rec casesplit vars dun pols cont sgns =
  match pols with
    [] -> matrix vars dun cont sgns
  | p::ops -> split_trichotomy sgns (head vars p)
                (if is_constant vars p then delconst vars dun p ops cont
                 else casesplit vars dun (behead vars p :: ops) cont)
                (if is_constant vars p then delconst vars dun p ops cont
                 else casesplit vars (dun@[p]) ops cont)
```

The `delconstant` function just removes the polynomial from the list and returns to case-splitting, except that it also modifies the continuation appropriately to put the sign back in the matrix before calling the original continuation:

```
and delconst vars dun p ops cont sgns =
  let cont' m = cont(map (insertat (length dun) (findsign sgns p)) m) in
  casesplit vars dun ops cont' sgns
```

Finally, we come to the main function `matrix`, where we assume that all the polynomials in the list `pols` are non-constant and have a head coefficient of

known nonzero sign. If the list of polynomials is empty, then trivially the empty sign matrix is the right answer, so we call the continuation on that. Note the exception trap, though! Because of our 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 — in which case `inferisign` will generate an exception. We don't actually want to fail here, but we're at liberty to return whatever formula we like, such as $\bot$.

Otherwise, we pick a polynomial `p` of maximal degree, so that we make definite progress in the recursive step: we remove at least one polynomial of maximal degree and replace it only with polynomials of lower degree, hence making the recursion wellfounded. We reshuffle the polynomials slightly to move it from position `i` to the head of the list, and add its derivative in front of that, giving `qs`. Then we form all the remainders `gs` from pseudo-division of `p` by each member of the `qs`, and recurse again on the new list of polynomials, starting with the case splits. The continuation is modified to apply `dedmatrix` and also to compensate for the shuffling of `p` to the head of the list:

```
and matrix vars pols cont sgns =
  if pols = [] then try cont [[]] with Failure _ -> False else
  let p = hd(sort(decreasing (degree vars)) 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 (pdivide_pos vars sgns p) qs in
  let cont' m = cont(map (fun l -> insertat i (hd l) (tl l)) m) in
  casesplit vars [] (qs@gs) (dedmatrix cont') sgns;;
```

To perform quantifier elimination from an existential formula, we first pick out all the polynomials (we assume atoms have already been normalized), set up the continuation to test the body on the resulting sign matrix, and call `casesplit` with the initial sign context.

```
let basic_real_qelim vars (Exists(x,p)) =
  let pols = atom_union
    (function (R(a,[t;Fn("0",[])])) -> [t] | _ -> []) p in
  let cont mat = if exists (fun m -> testform (zip pols m) p) mat
                 then True else False in
  casesplit (x::vars) [] pols cont init_sgns;;
```

Note that we can test *any* quantifier-free formula using the matrix, not just a conjunction of literals. So we may elect to do no logical normalization of the formula at all, certainly not a full DNF transformation. We will however evaluate and simplify all the time:

```
let real_qelim =
  simplify ** evalc **
  lift_qelim polyatom (simplify ** evalc) basic_real_qelim;;
```

Examples

We can try out the algorithm by testing if univariate polynomials have solutions:

```
# real_qelim <<exists x. x^4 + x^2 + 1 = 0>>;;
- : fol formula = <<false>>
# real_qelim <<exists x. x^3 - x^2 + x - 1 = 0>>;;
- : fol formula = <<true>>
```

and even, though not very efficiently, count them:

```
# real_qelim <<exists x y. x^3 - x^2 + x - 1 = 0 /\
                           y^3 - y^2 + y - 1 = 0 /\ ~(x = y)>>;;
- : fol formula = <<false>>
```

If you're still a bit puzzled by all the continuation-based code, you might find it instructive to *see* the sign matrix that gets passed to `testform`. One way is to switch on tracing; e.g. compare the output here with the example of a sign matrix we gave at the beginning:

```
# #trace testform;;
# real_qelim <<exists x. x^2 - 3 * x + 2 = 0 /\ 2 * x - 3 = 0>>;;
# #untrace testform;;
```

We can eliminate quantifiers however they are nested, e.g.

```
# real_qelim
   <<forall a f k. (forall e. k < e ==> f < a * e) ==> f <= a * k>>;;
- : fol formula = <<true>>
```

and we can obtain parametrized solutions to root existence questions, albeit not very compact ones:

```
# real_qelim <<exists x. a * x^2 + b * x + c = 0>>;;
- : fol formula =
<<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))>>
```

Moreover, we can check our own simplified condition by eliminating all quantifiers from a claimed equivalence, perhaps first guessing:

```
# real_qelim <<forall a b c. (exists x. a * x^2 + b * x + c = 0) <=>
                             b^2 >= 4 * a * c>>;;
- : fol formula = <<false>>
```

and then realizing we need to consider the degenerate case $a = 0$:

```
# real_qelim <<forall a b c. (exists x. a * x^2 + b * x + c = 0) <=>
                             a = 0 /\ (b = 0 ==> c = 0) \/
                             ~(a = 0) /\ b^2 >= 4 * a * c>>;;
- : fol formula = <<true>>
```

In section **??** we derived a canonical term rewriting system for groups, and we can prove that it is terminating using the following polynomial interpretation (Huet and Oppen 1980). To each term t in the language of groups we associate an integer value $v(t) > 1$, by assigning some arbitrary integer > 1 to each variable and then calculating the value of a composite term according to the following rules:

$$\begin{aligned} v(s \cdot t) &= v(s)(1 + 2v(t)) \\ v(i(t)) &= v(t)^2 \\ v(1) &= 2 \end{aligned}$$

We should first verify that this is indeed 'closed', i.e. that if $v(s)$ and $v(t)$ are both > 1, so are $v(s \cdot t)$, $v(i(t))$ and $v(1)$. (The other required property, being an integer, is preserved by addition and multiplication.) We can do this pretty quickly:

```
# real_qelim <<1 < 2 /\ (forall x. 1 < x ==> 1 < x^2) /\
               (forall x y. 1 < x /\ 1 < y ==> 1 < x * (1 + 2 * y))>>;;
- : fol formula = <<true>>
```

To avoid tedious manual transcription, we automatically translate terms to their corresponding "valuations", where the variables in a term are simply mapped to similarly-named variables in the value polynomial.

```
let rec grpterm tm =
  match tm with
    Fn("*",[s;t]) -> let t2 = Fn("*",[Fn("2",[]); grpterm t]) in
                     Fn("*",[grpterm s; Fn("+",[Fn("1",[]); t2])])
  | Fn("i",[t]) -> Fn("^",[grpterm t; Fn("2",[])])
  | Fn("1",[]) -> Fn("2",[])
  | Var x -> tm;;
```

Now to show that a set of equations $\{s_i = t_i \mid 1 \leq i \leq n\}$ terminates, it suffices to show that $v(s_i) > v(t_i)$ for each one. So let us map an equation $s = t$ to a new formula $v(s) > v(t)$, then generalize over all variables, relativized to reflect the assumptions that they are all > 1:

```
let grpform (Atom(R("=",[s;t]))) =
  let fm = generalize(Atom(R(">",[grpterm s; grpterm t]))) in
  relativize(fun x -> Atom(R(">",[Var x;Fn("1",[])]))) fm;;
```

After running completion to regenerate the set of equations:

```
let eqs = complete_and_simplify ["1"; "*"; "i"]
  [<<1 * x = x>>; <<i(x) * x = 1>>; <<(x * y) * z = x * y * z>>];;
```

Now we can create the critical formula and test it:

```
# let fm = list_conj (map grpform eqs);;
val fm : fol formula =
  <<(forall x4.
       x4 > 1 ==>
       (forall x5.
          x5 > 1 ==> (x4 * (1 + 2 * x5))^2 > x5^2 * (1 + 2 * x4^2))) /\
    (forall x1. x1 > 1 ==> x1^2^2 > x1) /\
    ...
   >>;;
# real_qelim fm;;
- : fol formula = true
```

Improvements

The decidability of the theory of reals is a remarkable and theoretically useful result. In principle, we could use `real_qelim` to settle unsolved problems such as finding kissing numbers for spheres in various dimensions (Conway and Sloane 1993). In practice, such a course is completely hopeless even for much simpler quantifier elimination problems like $\forall x.\, x^4 + px^2 + qx + r \geq 0$ (Lazard 1988). The bad theoretical complexity bounds (Vorobjov 1990) seem to be an insuperable practical obstacle. Motivated by the 'feeling that a single algorithm for the full elementary theory of $\mathbf{R}$ can hardly be practical' (Dries 1988), many authors have investigated special heuristic mixtures of algorithms for restricted subcases. We discuss a quite different approach to proving purely universal formulas over $\mathbb{R}$ in section **??** below.

One particularly notable failing of our algorithm is that it does not exploit equations in the initial problem to perform cancellation by pseudo-division, yet in many cases this would be a dramatic improvement — see exercise **??** below. Indeed, even Collins's original CAD algorithm, according to Loos and Weispfenning (1993), performed badly on the following:

$$\exists c.\ \forall b.\ \forall a.\ (a = d \wedge b = c) \vee (a = c \wedge b = 1) \Longrightarrow a^2 = b$$

We do poorly here too, but if we first split the formula up into DNF:

```
let real_qelim' =
  simplify ** evalc **
  lift_qelim polyatom (dnf ** cnnf (fun x -> x) ** evalc)
                      basic_real_qelim;;
```

the situation is much better:

```
# real_qelim'
  <<forall d.
      (exists c. forall a b. (a = d /\ b = c) \/ (a = c /\ b = 1)
                             ==> a^2 = b)
      <=> d^4 = 1>>;;
- : fol formula = <<true>>
```

A refinement of this idea of elimination using equations, developed and successfully applied by Weispfenning (1997), is to perform 'virtual term substitution' to replace other instances of x constrained by a polynomial $p(x) = 0$ by expressions for the roots of that polynomial. In the purely linear case, things are better still. Given an existentially quantied conjunction, we can eliminate variables completely using substitution with any linear equations. For inequalities we can use a slight elaboration of the DLO procedure to eliminate variables based on:

$$(\exists x.\ (\bigwedge_i s_i < x) \wedge (\bigwedge_j x < t_j)) \Leftrightarrow \bigwedge_{i,j} s_i < t_j$$

However, even this can have exponential worst-case complexity, because each variable elimination can roughly square the number of inequalities. For large problems, one might turn instead to traditional linear programming techniques. The classic simplex method (Dantzig 1963) often works well in practice, and more recent interior-point algorithms following Karmarkar (1984) even have provable polynomial-time bounds.[4]

[4] This only implies polynomial time solvability for quantifier elimination problems without quantifier alternations. The linear programming problem was famously proved to be solvable in polynomial time by Khachian (1979), using a reduction to approximate convex optimization, solvable in polynomial time using the ellispoid algorithm. However, the implicit algorithm was seldom competitive with simplex in practice. See Grotschel, Lovsz, and Schrijver (1993) for a detailed discussion of the ellipsoid algorithm and its remarkable generality.

Bibliography

[Caviness and Johnson (1998]Caviness, B. F. and Johnson, J. R. (eds.) (1998) *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and monographs in symbolic computation. Springer-Verlag.

[Cohen (1969]Cohen, P. J. (1969) Decision procedures for real and p-adic fields. *Communications in Pure and Applied Mathematics*, **22**, 131–151.

[Collins (1976]Collins, G. E. (1976) Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In Brakhage, H. (ed.), *Second GI Conference on Automata Theory and Formal Languages*, Volume 33 of *Lecture Notes in Computer Science*, Kaiserslautern, pp. 134–183. Springer-Verlag.

[Conway and Sloane (1993]Conway, J. H. and Sloane, N. J. A. (1993) The kissing number problem. In *Sphere Packings, Lattices, and Groups* (second ed.)., pp. 21–24. Springer-Verlag.

[Dantzig (1963]Dantzig, G. B. (1963) *Linear Programming and Extensions*. Princeton University Press.

[Dries (1988]Dries, L. v. d. (1988) Alfred Tarski's elimination theory for real closed fields. *Journal of Symbolic Logic*, **53**, 7–19.

[Grotschel, Lovsz, and Schrijver (1993]Grotschel, M., Lovsz, L., and Schrijver, A. (1993) *Geometric algorithms and combinatorial optimization*. Springer-Verlag.

[Hörmander (1983]Hörmander, L. (1983) *The Analysis of Linear Partial Differential Operators II*, Volume 257 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag.

[Huet and Oppen (1980]Huet, G. and Oppen, D. C. (1980) Equations and rewrite rules: a survey. In Book, R. V. (ed.), *Formal Language Theory: Perspectives and Open Problems*, pp. 349–405. Academic Press.

[Karmarkar (1984]Karmarkar, N. (1984) A new polynomial-time algorithm for linear programming. *Combinatorica*, **4**, 373–395.

[Khachian (1979]Khachian, L. G. (1979) A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, **20**, 191–194.

[Kreisel and Krivine (1971]Kreisel, G. and Krivine, J.-L. (1971) *Elements of mathematical logic: model theory* (Revised second ed.). Studies in Logic and the Foundations of Mathematics. North-Holland. First edition 1967. Translation of the French 'Eléments de logique mathématique, théorie des modeles' published by Dunod, Paris in 1964.

[Lazard (1988]Lazard, D. (1988) Quantifier elimination: Optimal solution for two classical examples. *Journal of Symbolic Computation*, **5**, 261–266.

[Łojasiewicz (1964]Łojasiewicz, S. (1964) Triangulations of semi-analytic sets. *Annali della Scuola Normale Superiore di Pisa, ser. 3*, **18**, 449–474.

[Loos and Weispfenning (1993]Loos, R. and Weispfenning, V. (1993) Applying linear quantifier elimination. *The Computer Journal*, **36**, 450–462.

[Rabin (1991]Rabin, M. O. (1991) Decidable theories. In Barwise, J. and Keisler, H. (eds.), *Handbook of mathematical logic*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*, pp. 595–629. North-Holland.

[Seidenberg (1954]Seidenberg, A. (1954) A new decision method for elementary algebra. *Annals of Mathematics*, **60**, 365–374.

[Sturm (1835]Sturm, C. (1835) Mémoire sue la résolution des équations numériques. *Mémoire des Savants Etrangers*, **6**, 271–318.

[Tarski (1951]Tarski, A. (1951) *A Decision Method for Elementary Algebra and Geometry.* University of California Press. Previous version published as a technical report by the RAND Corporation, 1948; prepared for publication by J. C. C. McKinsey. Reprinted in Caviness and Johnson (1998), pp. 24–84.

[Vorobjov (1990]Vorobjov, N. N. (1990) Deciding consistency of systems of polynomial in exponent inequalities in subexponential time. In Mora, T. and Traverso, C. (eds.), *Proceedings of the MEGA-90 Symposium on Effective Methods in Algebraic Geometry*, Volume 94 of *Progress in Mathematics*, Castiglioncello, Livorno, Italy, pp. 491–500. Birkhäuser.

[Weispfenning (1997]Weispfenning, V. (1997) Quantifier elimination for real algebra — the quadratic case and beyond. *Applicable Algebra in Engineering Communications and Computing*, **8**, 85–101.