Linear and Nonlinear Arithmetic in ACL2

Warren A. Hunt, Jr., Robert Bellarmine Krug, and J Moore

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
{hunt,rkrug,moore}@cs.utexas.edu

Abstract. As of version 2.7, the ACL2 theorem prover has been extended to automatically verify sets of polynomial inequalities that include nonlinear relationships. In this paper we describe our mechanization of linear and nonlinear arithmetic in ACL2. The nonlinear arithmetic procedure operates in cooperation with the pre-existing ACL2 linear arithmetic decision procedure. It extends what can be automatically verified with ACL2, thereby eliminating the need for certain types of rules in ACL2's database while simultaneously increasing the performance of the ACL2 system when verifying arithmetic conjectures. The resulting system lessens the human effort required to construct a large arithmetic proof by reducing the number of intermediate lemmas that must be proven to verify a desired theorem.

1 Introduction

Mechanical theorem proving or proof checking systems offer a rigorous methodology with which to structure and check proofs. Each such system offers a different degree of automation – directly affecting its capability and ease of use. We have extended the ACL2 theorem proving system [7,8,9] with an automated verification procedure that enhances the linear arithmetic decision procedure. ACL2 can now more easily verify sets of inequalities containing nonlinear arithmetic relationships.

In this paper we describe our mechanization of linear and nonlinear arithmetic in ACL2. Before doing so, we briefly describe the theory behind the procedures and provide a couple of trivial examples of their use. The procedures operate on inequalities over the rationals. These inequalities can be combined by cross-multiplication and addition to permit the deduction of an additional inequality. For example, if $0 < poly1$ and $0 < poly2$, and c and d are positive rational constants, then $0 < c \cdot poly1 + d \cdot poly2$. Here, we are use two facts: multiplication by a positive rational constant does not change the sign of a polynomial and the sum of two positive polynomials is itself positive. This is linear arithmetic. We also have that $0 < c \cdot poly1 \cdot poly2$. In this nonlinear case, we are using the fact that the product of two positive polynomials is itself positive.

Now suppose we want to prove

$$3 \cdot x + 7 \cdot a < 4 \quad \wedge \quad 3 < 2 \cdot x \quad \Longrightarrow \quad a < 0.$$

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 319–333, 2003.

To do this, we assume the two hypotheses and the negation of the conclusion, and look for a contradiction. We therefore start with the three inequalities:

$$0 < -3 \cdot x + -7 \cdot a + 4 \tag{1}$$

$$0 < 2 \cdot x + -3 \tag{2}$$

$$0 \leq a. \tag{3}$$

We cross-multiply and add the first two – that is, multiply inequality (1) by two and inequality (2) by three, and then add the respective sides. This yields

$$0 < -14 \cdot a + -1. \tag{4}$$

Note that the new inequality does not mention x. If we choose two inequalities with the same leading term and leading coefficients of opposite sign, we can generate an inequality in which that leading term is not present. This is the general strategy employed by the linear arithmetic decision procedure.

If we next cross-multiply and add inequality (4) with inequality (3), we get

$$0 < -1, \tag{5}$$

a false polynomial. We have, therefore, proved our theorem.

This process illustrated above of cross-multiplying and adding two inequalities will be referred to as "cancelling" the two inequalities. We shall refer to such obviously false inequalities as (5) as "contradictions," and speak of any process that results in one of these as "generating a contradiction."

Next, suppose that we have the three assumptions

$$3 \cdot x \cdot y + 7 \cdot a < 4 \quad \text{or} \quad 0 < -3 \cdot x \cdot y + -7 \cdot a + 4 \tag{6}$$

$$3 < 2 \cdot x \quad \text{or} \quad 0 < 2 \cdot x + -3 \tag{7}$$

$$1 < y \quad \text{or} \quad 0 < y + -1, \tag{8}$$

and we wish to prove that $a < 0$. We proceed by assuming the negation of our goal, $0 <= a$, and looking for a contradiction.

Note that in this case no two inequalities have a leading term in common. In this situation there are no cancellations to perform. However, (6) has a product as its leading term, $x \cdot y$, and for each of the factors of that product, x and y, there is an inequality which has such a factor as a leading term. When nonlinear arithmetic is enabled, ACL2 will multiply (7) and (8), obtaining

$$0 < 2 \cdot x \cdot y + -3 \cdot y + -2 \cdot x + 3. \tag{9}$$

The addition of this polynomial will allow cancellation to continue[1] and, in this case, we will prove our goal. Thus, just as ACL2 adds two polynomials when they have the same largest unknown of opposite signs in order to create a new smaller polynomial, ACL2 can now multiply polynomials when the product of their largest unknowns is itself the largest unknown of another polynomial.

[1] Inequality 9 can be canceled with 6. The result can be canceled with 8, and so on. The final cancellation will be with the negation of our goal, $0 <= a$.

1.1 Related Work and Plan of the Paper

It is often desirable to verify the correct operation of computer hardware or software. These operations may involve arithmetic, as in the floating-point hardware of a modern microprocessor or pointer arithmetic in a C program.

Several approaches to automating the verification of arithmetic lemmas have been tried. Great progress has been made as is illustrated by the many substantial proofs recently completed in PVS, HOL, and ACL2 [10,5,13]. The existing state-of-the-art is, however, not sufficient. The level of user expertise and effort required for the above-mentioned work is too high.

One of the primary difficulties encountered has been the fact that the formulae to be proved are rarely limited to just the four basic arithmetic operations, $+$, $-$, $*$, and $/$, but often involve diverse semantic constructs or, at the least, user-defined functions. Theorem provers, therefore, cannot limit themselves to "pure" arithmetic but must work with combinations of theories.

Our approach has developed from an engineering, results-oriented perspective, and we have therefore concentrated on decreasing the user's effort for the types of lemmas we see ACL2 users attempting to prove. Others have taken a more theoretical approach, whereby they can guarantee algorithmic completeness[2] over an exactly specified domain.

Several groups have built such systems by combining small-domain specific provers. Nelson and Oppen [11] and Shostak [14] describe frameworks with which one can combine separate existing decision procedures into one larger procedure. This work has been extended by others; e.g., Kapur [6] and SRI [12]. While this approach has some nice properties, such as completeness and efficiency, it can be somewhat limiting. Some of these limitations arise because the procedures to be integrated are treated as fixed black-boxes. Armando and Ranise [1] describe a method for augmenting the black-boxes. Other limitations arise from concerns over efficiency. Harrison [4] explores the use of a full decision procedure over the reals and discusses its desirability.

We build on earlier work by Boyer and Moore [2] and share a common design philosophy with theirs. We regard ACL2's various procedures as a mutually recursive nest of functions, and have tuned both the interfaces and internals of these procedures using feedback from users to guide the process. It is also similar to work by Cyrluk and Kapur [3]; they too were concerned with augmenting existing linear arithmetic decision procedures to handle nonlinear inequalities, and their design was also driven by engineering rather than theoretical concerns. We have the advantage of possessing much faster computers than were available at that time and believe that the time has come to reexamine the feasibility of more ambitious, but still incomplete algorithms, for handling nonlinear arith-

[2] A decision procedure is said to be complete if it always returns a (correct) answer when asked to verify a true theorem. An incomplete procedure, by contrast, may return an "I don't know" or even not return at all.

metic[3]. In this paper we present our first attempt at fully integrating a nonlinear arithmetic semi-decision procedure into ACL2. We present merely an outline of our work, and do not discuss nor even mention many of the heuristics that we have employed to limit and guide our algorithms.

We provide the required background including the definitions of polys, pots, and labels, as well as a short discussion of type reasoning and linearization. Thereafter we describe the subprocedures that make up the linear and nonlinear arithmetic procedures. The linear arithmetic procedure consists of two nested loops. The innermost of these, the linear arithmetic decision procedure (described in Section 2) is responsible for adding inequalities to the pot-list. In Section 3 we present linear arithmetic's outer loop, the linear lemmas procedure which attempts to gather additional inequalities in order to allow further cancellations. The nonlinear arithmetic procedure consists of three nested loops. The innermost is the same as for linear arithmetic. The next, described in Section 4, is an augmentation of that described in Section 3. Nonlinear arithmetic's outer loop is presented in Section 5. We conclude with a few remarks about the labor that can now be saved.

1.2 Polys, Pots, Pot-lists, and Labels

The procedures we will describe here operate on polynomial inequalities over the rationals. A "polynomial" is a sum of terms, each of which is either a rational constant or the product of a rational constant and an "unknown." An example polynomial is $3 \cdot x + -7/2 \cdot a + 2$; here x and a are the unknowns. The unknowns, however, need not be variable symbols; e.g., $|x|$, x^n, or $f(x, y)$ may be used as unknowns. Thus, $-3 \cdot |x| + a$ is also a polynomial.

A "polynomial inequality," or a "poly" for short, is an inequality (either $<$ or $\leq$) between 0 and a polynomial; e.g., $0 < 3 \cdot x + -7/2 \cdot a + 2$ and $0 \leq -3 \cdot |x| + a$ are polys. We refer to obviously false polys such as $0 < 0$ as "contradictions."

Polys are stored in groups called "pots." All the polys with the same largest unknown[4] are stored in a single pot, which is said to be "labeled by" or "about" that unknown. These pots are further divided into two compartments – one for "positive" polys (with a positive leading coefficient) and one for "negative" polys (with a negative leading coefficient). A pot represents the conjunction of the polys in it.

The pots are stored in a "pot-list," which represents the conjunction of the pots in it. An example[5] is:

[3] One simple example of incompleteness is our inability to (automatically) prove $x \cdot x \neq 2$. If we were operating over the reals, $x = \sqrt{2}$ would be a solution, but recall that we are operating over the rationals. The authors have not done a study of how to demarcate the class of formulas on which our algorithms succeed or fail. They are, however, unaware of any examples which "should" be proveable using these techniques but which are not proveable for reasons other than limiting heuristics.

[4] The order used here is basically lexicographic, considering number of variables first, number of function symbols second, and alphabetical order last.

[5] Note that there are cancellations which can be performed.

label	positives	negatives
b	$0 \leq b + -1 \cdot a + -1$	
$f(x,\, y)$	$0 \leq f(x,\, y)$	$0 < -1 \cdot f(x,\, y) + -1 \cdot b + a$

We refer to the b and $f(x,\, y)$ above as their pots' "labels."

The procedures that we will describe here all take, among other arguments, a pot-list and a list of polys to be added to the pot-list. They return either an augmented pot-list or a contradiction (a false poly) – the latter case indicating success as in Section 1.

1.3 Type Reasoning and Polys from Type-Set

We shall treat type reasoning – carried out by calling the ACL2 function `type-set` – as something of a black-box. For present purposes only a few things need be known about it.

First, we use it here to quickly answer the question "To what arithmetic category does this expression belong?" where the possible answers are zero,[6] positive integer, negative integer, positive ratio, negative ratio, or combinations thereof such as non-negative rational.

Second, we can sometimes form polys about an expression based on the answer given. For example, if x is said to be a nonpositive rational, we can create the poly $0 \leq -1 \cdot x$ from that information. We refer to this mechanism as "creating polys from `type-set`."

Third, although `type-set`'s reasoning abilities are fairly limited, they can be extended through the use of type-prescription rules. ACL2 comes with some of these already built in including rules about the basic arithmetic functions such as that $x \cdot y$ is a positive rational if both x and y are. This rule can, via the above-mentioned mechanism of creating polys from `type-set`, provide a small amount of nonlinear reasoning to the linear arithmetic procedures. We shall see this shortly.

1.4 Linearization

Linearization is the process of converting an ACL2 term into one or more polys. We note the following:

1. An equality can be expressed as a conjunction of two inequalities; $x = y$ is true if and only if both $x \leq y$ and $y \leq x$ are true.[7].
2. We normalize polys so that their leading coefficient is $+/-1$.
3. Consider the ACL2 term[8] `(< x y)`. If we know that both `x` and `y` are integers, we can assume that we are linearizing `(<= (+ x 1) y)` instead, and

[6] A category with only one member.

[7] Note that the negation of an equality can similarly be expressed as a *disjunction* of two inequalities. We do not address this further in the present paper except to say that ACL2 does handle such situations.

[8] ACL2 terms are a subset of Lisp expressions, and therefore use a Lisp-style prefix syntax.

so convert `(< x y)` to the poly $0 \leq y + -1 \cdot x + -1$ rather than the weaker $0 < y + -1 \cdot x$. We shall refer to this as "the 1+ trick." This is the only place in which the procedures described here take advantage of the discreteness of the integers.

2 Linear Arithmetic

In this section, we describe the innermost loop of ACL2's arithmetic procedures. The algorithm described in this section is a decision procedure for linear arithmetic over the rationals. We later refer to this as the linear arithmetic algorithm.

As in the examples of Section 1, our goal is to derive a contradiction. In order to do so, all of the unknowns of a poly must be eliminated by cancellation. We can choose to eliminate them in any order, but we eliminate the first. That is, two polys are canceled against each other only when their largest unknowns are the same and have coefficients of opposite signs. Note that this occurs precisely when two polys are (or will be) in opposite sides of the same pot.

2.1 The Linear Arithmetic Algorithm

We start with a (possibly empty) pot-list and a list of polys to be added to it. We repeat the following until we reach a fixed point.

1. For each poly to be added:
   find its pot (the one whose label matches the poly's largest unknown), if there is one, or make a new one. Add the poly to this pot and cancel the new poly with any polys of the opposite sign. If this generates a contradiction, quit and return the contradiction; otherwise set any new polys aside.
2. If there were any new polys set aside in step 1, go back to step 1 with the new polys. Otherwise, go on to step 3.
3. For each pot that is new or has been changed by having polys added to it in step 1, try to create a poly from `type-set` about the label of the changed pot. Collect any such newly created polys and return to step 1 with them.

2.2 An Example

Suppose that we want to prove

$$\text{integer } a \quad \wedge \quad \text{integer } b \quad \wedge \quad 0 \leq a \quad \wedge \quad a < b \quad \Longrightarrow \quad a + 1 \leq a \cdot b + b$$

As before, we assume the hypotheses and the negation of the conclusion, and look for a contradiction. Since a and b are assumed to be integers, the linearization of $a < b$ is $0 \leq b + -1 \cdot a + -1$. Similarly $a \cdot b$ is known to be an integer since a and b are, so the linearization of the negation of $a + 1 \leq a \cdot b + b$ is $0 < -1 \cdot a \cdot b + -1 \cdot b + a$. Both of these linearizations used the 1+ trick. Finally, the linearization of $0 \leq a$ is just $0 \leq a$.

Since no cancellations can be performed between these three polys, executing steps 1 and 2 above results in the pot-list

label	positives	negatives
a	$0 \leq a$	
b	$0 \leq b + -1 \cdot a + -1$	
$a \cdot b$		$0 < -1 \cdot a \cdot b + -1 \cdot b + a.$

In step 3 we create three polys from `type-set`. There are three new (and therefore changed) pots, a, b, and $a \cdot b$. `Type-set` knows that a is a nonnegative integer and that b is a positive integer[9] and so we create the polys $0 \leq a$ and $0 < b$. As mentioned in 1.3, `type-set` therefore also knows that $a \cdot b$ is a nonnegative integer, and so we create the poly $0 \leq a \cdot b$. This small amount of nonlinear reasoning has long been built into ACL2. Note that we used the pot labels to guide our search for additional polys.

When we add these to the pot-list, after executing step 1 *once,* we get

label	positives	negatives
a	$0 \leq a$	
b	$0 < b$	
	$0 \leq b + -1 \cdot a + -1$	
$a \cdot b$	$0 \leq a \cdot b$	$0 < -1 \cdot a \cdot b + -1 \cdot b + a$

with the poly $0 < -1 \cdot b + a$ having been set aside. This poly is the result of cancelling the two polys in the $a \cdot b$ pot.[10] Upon adding it and canceling the polys in the b pot (executing steps 2 and 1 again), we get the contradiction $0 < -1$ and our lemma is proved.

3 Linear Lemmas

Prior to version 2.7, ACL2's arithmetic procedure encompassed little more than is described in this section. It is still the standard behavior of ACL2 when nonlinear arithmetic is disabled. Note that this procedure is not complete.

Suppose that the procedure described above does not produce a contradiction but instead yields a set of nontrivial polys. A contradiction might still be generated if we could add to the set some additional polys which allow further cancellation. That is where linear lemmas come in. Linear lemmas are more general and powerful than polys from type-set. (An example follows shortly.) When the set of polys has stabilized under the procedure described above and no contradiction has been produced, we form a list of the labels of any newly created pots and search the database of linear rules for ones that pattern match with a pot label. For each rule found, if we are able to relieve its hypotheses, we add its

[9] The variable a is nonnegative by hypothesis, and since b is strictly greater than a, b must be positive. This is about as complicated as type-reasoning gets.

[10] Note that cancellation does not remove any polys. We augment, but never diminish, the pot-list.

conclusion to the pot-list (using the above linear arithmetic algorithm) in the hope that this will allow further cancellations to proceed. Just as for polys from `type-set`, we are using pot labels to guide our search for additional polys. Such labels, recall, correspond to the unknowns that are candidates for cancellation.

3.1 The Linear Lemmas Algorithm

As before, we start with a list of polys and a (possibly empty) pot list. We repeat the following until we reach a fixed point or are interrupted by the user aborting the proof attempt.

1. Add the polys with the linear arithmetic algorithm as described in section 2.1; if no contradiction was generated, go on to step 2.
2. Make a list of the labels from any new pots created in step 1 (or 3). If there aren't any, quit and return the pot-list; otherwise, go on to step 3.
3. For each item in this list and for each applicable linear-lemma:
   If we can relieve the lemma's hypotheses, add the concluding poly(s) to the pot-list as described in section 2.1.
4. If a contradiction was generated, quit and return it. Otherwise, go back to step 2.

3.2 An Example

Suppose that we are given the following linear lemma,[11] `expt-lemma`, about x^n

$$1 < x \quad \wedge \quad \text{integer } n \quad \wedge \quad 1 < n \quad \Longrightarrow \quad x < x^n,$$

and that we wish to prove

$$2 < x \quad \wedge \quad \text{integer } n \quad \wedge \quad 1 < n \quad \wedge \quad a \leq x + b \quad \Longrightarrow \quad a < x^n + b.$$

After linearizing the inequalities among the hypotheses and the negation of the conclusion and adding them to the empty pot-list (step 1) we get

label	**positives**	**negatives**
b		$0 < -1 \cdot b + a$
n	$1 < n$	
x	$0 \leq x + b + -1 \cdot a$	
	$0 < x + -2$	
x^n	$0 < x^n$	$0 \leq -1 \cdot x^n + -1 \cdot b + a$

Note that the poly $0 < x^n$ was created from `type-set` about the pot-label x^n.

[11] Note that the conclusion of this lemma does not encode a type such as positive integer, and so could not be expressed as a type-prescription rule. We also mention here that the hypotheses of a linear lemma may be relieved by general purpose rewriting and (recursively) linear and nonlinear arithmetic, while a type-prescription rule's hypotheses must be relieved by type-reasoning only.

In step 2, we note that there were four new pots created in step 1, and in step 3 we will eventually find `expt-lemma`. We sketch here how we relieve the first hypothesis, $1 < x$. Rewriting cannot do anything with this, so we linearize the negation of the hypothesis (yielding $0 \leq -1 \cdot x + 1$) and recursively call the very procedure we are describing. In this situation we do not start with an empty pot-list. This poly will be added to the x pot. Upon cancellation with $0 < x + -2$, we get the contradiction $0 < -1$, and the hypothesis has been relieved.

We therefore add the linearization of the conclusion of `expt-lemma`, $0 < x^n + -1 \cdot x$, to the pot-list. After a couple of rounds of cancellation we derive the contradiction $0 < -2$, and the theorem has been proved.

4 Linear Lemmas Revised

When nonlinear arithmetic is enabled, we do the above procedure a little differently. The gathering of polys from linear lemmas is intended to let the process of cancellation continue. In the procedure described in this section we still use linear lemmas, but we intertwine their use with other ways of gathering polys in preparation for what is to come – the nonlinear arithmetic procedure.

4.1 Exploded Pot Labels, Bounds Polys, and Inverse Polys

Previously, we examined pot labels to direct our gathering of additional polys from such sources as `type-set` and linear lemmas. That is, when there was a pot labeled with, say, x^n, we looked to `type-set` or linear lemmas for additional information about x^n. We shall soon examine "exploded" pot labels. These exploded pot labels consist of the original pot label and, if the pot label is a product, each of the label's factors. A few examples will make this clearer:

- $x \implies x$
- $\lfloor x \rfloor \implies \lfloor x \rfloor$
- $x \cdot \lfloor x \rfloor \implies x$, $\lfloor x \rfloor$, and $x \cdot \lfloor x \rfloor$
- $x \cdot y \cdot z \implies x$, y, z, and $x \cdot y \cdot z$

We are doing this so that we can seed the database with information about the factors of products. Note that in the last example we do not examine, for instance, $x \cdot y$.

A "bounds poly" is a poly with exactly one unknown and can be considered to bound the unknown. For instance, $0 < x + 1$ can be considered to give a lower bound of -1 for x. Similarly, $0 < -1 \cdot x + 3$ bounds x from above at 3. A term is said to have "good bounds" if there are bounds polys for that term which bound the term away from zero. This will become important later when we multiply certain polys. For example, we may wish to multiply, $0 < x \cdot \frac{1}{y}$ and $0 < y$ to form the new poly $0 < x$. But, since this requires rewriting $y \cdot \frac{1}{y}$ to 1, it can be done only if y is known to be non-zero.

Division thus introduces additional issues. We represent the ratio x/y as $x \cdot \frac{1}{y}$. A term is said to "involve division" if it is of the form

1. $\frac{1}{x}$ or $(\frac{1}{x})^n$, or
2. x^c or $x^{c \cdot n}$, where c is a constant negative integer.

As preparation for the nonlinear procedure, given such a term about division, ACL2 "adds its inverse polys." We do not attempt to describe the method for generating these polys here other than to say that we gather our initial information from the bounds polys present in the pot-list. We give a few examples:

- If we can determine that $4 < x$, we know both $0 < \frac{1}{x}$ and $\frac{1}{x} < 1/4$.
- If we can determine that $0 < \frac{1}{x}$ and $\frac{1}{x} < 3$, we know $1/3 < x$.
- If we can only determine that $-2 < x$, we do not know anything about $\frac{1}{x}$.

4.2 The Revised Linear Lemmas Algorithm

As mentioned above, when nonlinear arithmetic is enabled we do things a bit differently. As before, we start with a list of polys and a (possibly empty) pot list. We repeat the following until we reach a fixed point or are interrupted by the user aborting the proof attempt.

1. Add the polys with the linear arithmetic algorithm as described in section 2.1; if no contradiction was generated, go on to step 2.
2. Make an exploded list of the labels of any new pots. If there are not any, quit and return the new pot-list. Otherwise, go on to step 3.
3. For each item in this list:
   a) Add any polys created from `type-set`.
   b) For each applicable linear lemma: If we can relieve its hypotheses, add the concluding poly(s) to the pot-list as in section 2.1.
   c) If the item involves division, add any inverse polys
4. If a contradiction was generated, quit and return it. Otherwise, go back to step 2.

4.3 An Example – Part I

Let us consider $0 < a \quad \wedge \quad a < b \quad \Longrightarrow 1 < b/a$. After adding the initial polys, the pot-list will look like

label	**positives**	**negatives**
a	$0 < a$	
b	$0 < b + -1 \cdot a$	
$b \cdot \frac{1}{a}$		$0 \leq -1 \cdot b \cdot \frac{1}{a} + 1$

In step 2, we make the list a, b, $\frac{1}{a}$, $b \cdot \frac{1}{a}$. Note the presence of $\frac{1}{a}$, which would not have been there if we used regular pot labels. In step 3a we create the poly $0 < \frac{1}{a}$, among others, from `type-set` and add it to the pot.[12] We will continue this example in Section 5.1.

[12] We also create the same poly in step 3c.

5 Nonlinear Arithmetic

Before proceeding, let us pause a moment to recollect where we are. In Section 2.1, we presented the linear arithmetic algorithm which lies at the heart of ACL2's arithmetic procedures – both linear and nonlinear. We next described the previously existing linear lemmas algorithm in Section 3.1. This algorithm uses the linear arithmetic algorithm and is still the default behaviour when nonlinear arithmetic is not enabled. Next, in Section 4.2 we described a variant of the linear lemmas algorithm which is used when nonlinear arithmetic is enabled. Whereas, when nonlinear arithmetic is disabled, the previously existing linear lemmas algorithm is the outermost loop for arithmetic reasoning; the new variant is only the middle loop when nonlinear arithmetic is enabled. We are now about to describe the outermost loop of the nonlinear arithmetic algorithm.

The nonlinear arithmetic procedure consists of three subprocedures: deal-with-product, deal-with-factor, and deal-with-division. Each of these subprocedures is guided by pot-labels and attempts to multiply polys. In order to multiply two polys, we unlinearize the polys (converting them back into ACL2 terms), create the term representing their product, use general-purpose rewriting to rewrite the product terms, and linearize the result. For example, the product of the two polys $0 < -1 \cdot x + 3$ and $0 \leq y + a$ is $0 \leq -1 \cdot y \cdot x + -1 \cdot x \cdot a + 3 \cdot y + 3 \cdot a$. In order to multiply two pots, form a list of the polys in each pot and multiply each poly in the first list with each poly in the second. We multiply more than two polys or pots by generalizing the above.

5.1 Deal-with-Product and Deal-with-Factor

When we have polys about a product and we have polys about the product's factors, we can multiply those polys about the factors to form polys about the product and perhaps thereby allow cancellation to proceed.

For instance, if we have a new pot about the product $a \cdot b \cdot c$, we can form new polys about the product by finding pots with any of the following combinations of labels and then multiplying the pots.

- a, b, and c
- a, and $b \cdot c$
- $a \cdot c$, and b
- $a \cdot b$, and c

This is done by the subprocedure deal-with-product.

Similarly, if the new pot is about a, we look for pots of which a is a factor, such as $a \cdot b \cdot c$, and then see if we can complete the product. This is done by the subprocedure deal-with-factor. We use these two procedures in tandem so that we are less sensitive to the order in which pots are created.

Let us revisit the example from 4.2. When we left it, having just added the poly from `type-set`, it looked like

label	positives	negatives
a	$0 < a$	
$\frac{1}{a}$	$0 < \frac{1}{a}$	
b	$0 < b$	
	$0 < b + -1 \cdot a$	
$b \cdot \frac{1}{a}$	$0 < b \cdot \frac{1}{a}$	$0 \leq -1 \cdot b \cdot \frac{1}{a} + 1$

Both deal-with-product and deal-with-factor take a pot-label to consider and a pot-list. Deal-with-product will be used only with products, such as $b \cdot \frac{1}{a}$ above, while deal-with-factor will be used only with individual factors, such as a, b, and $\frac{1}{a}$ above.

When deal-with-product is given $b \cdot \frac{1}{a}$, it will find the pots for b and $\frac{1}{a}$ and multiply the two pots. In particular, it will multiply the polys $0 < \frac{1}{a}$ and $0 < b + -1 \cdot a$ getting $0 < b \cdot \frac{1}{a} + -1 \cdot a \cdot \frac{1}{a}$. This will be rewritten to

$$0 < b \cdot \frac{1}{a} + -1$$

since a is known to be non-zero. Upon adding this to the pot-list we would get the contradiction $0 < 0$ and be done.

When deal-with-factor is given a, it will do nothing because a is not a factor of any pot-labels. However, when it is given b, it will find the product $b \cdot \frac{1}{a}$. The pot-label $\frac{1}{a}$ will complete this product, and so deal-with-factor will multiply the pots for b and $\frac{1}{a}$ with the same results as for deal-with-product. The pot label $\frac{1}{a}$ is delt with similar.

5.2 Deal-with-Division

Let us next consider $0 < b \quad \wedge \quad b < a \quad \Longrightarrow \quad 1 < a/b$. After executing the revised linear lemmas algorithm, the pot-list will look like

label	positives	negatives
a	$0 < a$	
b	$0 < b$	$0 < -1 \cdot b + a$
$\frac{1}{b}$	$0 < \frac{1}{b}$	
$a \cdot \frac{1}{b}$	$0 < a \cdot \frac{1}{b}$	$0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$

This time, deal-with-product and deal-with-factor are insufficient. If we multiply $0 < a$ and $0 < \frac{1}{b}$, we get $0 < a \cdot \frac{1}{b}$, which we already knew via polys from type-set. Rather, we want to multiply the polys $0 < b$ and $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$. After rewriting $a \cdot b \cdot \frac{1}{b}$ to a, we have $0 < b + -1 \cdot a$. Upon adding this latter poly to the pot-list, we get $0 < 0$ and the lemma is proved.

We now sketch the algorithm behind deal-with-division.

1. If the current pot label being considered is itself a product, quit. If we have good bounds for the label, go to step 2; if not, quit.
2. Make a list of all the pot labels that have the multiplicative inverse of the current label as a factor. To distinguish them from the current pot, we will refer to the pots these labels belong to as the "found" pots. For each entry in this list:

a) Multiply the bounds polys from the current pot and the polys in the found pot.
b) Multiply the bounds polys from the found pot and the bounds polys from the current pot.

Let us see how this lines up with what we said we wanted to do above. When deal-with-division is examining the pot-label b (which has good-bounds) it finds the pot-labels $\frac{1}{b}$ and $a \cdot \frac{1}{b}$. For the second of these, since $0 < b$ and $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$ are both bounds polys, we multiply these polys in step 2. As above, upon adding this latter poly to the pot-list, we get $0 < 0$ and the lemma is proved.

5.3 The Nonlinear Arithmetic Algorithm

After adding polys as in Section 2.1, loop through the following at most three times. If at any point we generate a contradiction, quit and return it.

1. Execute the revised linear lemmas algorithm, described in Section 4.2.
2. Make a list (not an exploded list) of the labels from any new pots and for each item in that list:
   a) If we have good-bounds for the current item, carry out deal-with-division and add any polys generated.
   b) If the current item is a product, carry out deal-with-product and add any polys generated.
   c) If the current item is not a product, carry out deal-with-factor and add any polys generated.

This concludes our presentation of ACL2's nonlinear arithmetic algorithm.

6 Conclusion

The nonlinear arithmetic procedure is tightly integrated with the rest of ACL2 and allows lemmas such as the following to be proven automatically.

- This lemma was needed for an industrial project to verify the correctness of a microprocessor. It inspired our original work on nonlinear arithmetic and was an early success.

$$\begin{gathered} e < a \quad \wedge \quad a \leq d \quad \wedge \quad i < h \quad \wedge \quad h < g \quad \wedge \quad g \leq f \\ a \cdot f - a \cdot h \leq b + c \cdot g - c \cdot h \\ \Longrightarrow e \cdot f - e \cdot i \leq b + c \cdot g - c \cdot h + d \cdot h - d \cdot i \end{gathered}$$

- Proving this equality helped us to refine deal-with-division.

$$(\mathrm{bc}\, i\ j) = \frac{i!}{j!(i-j)!}$$

Where `bc` is the binomial coefficient defined by Pascal's Triangle:

```
(bc i \; j) =
if i < 0 or i < j then return 0
   elseif j <= 0 then return 1
   else return (bc i-- j) + (bc i-- j--)
```

- This was a long-standing challenge problem given to us by our sponsors. Previous versions of this proof required a dozen or more helper lemmas to be proven; we can now do the proof with only the one helper lemma given below.
  Consider the following 6502 assembly program to multiply two 8-bit numbers:

```
                    ; Multiply F1 and F2, leaving 16 bit
                    ; result in A and LOW
        LDX #8      ; Load X immediate with the integer 8
        LDA #0      ; Load A immediate with the integer 0
LOOP    ROR F1      ; Rotate F1 right circular through C
        BCC ZCOEF   ; Branch to ZCOEF if C = 0
        CLC         ; Set C to 0
        ADC F2      ; Set A to A+F2+c and C to the carry
ZCOEF   ROR A       ; Rotate A right circular through C
        ROR LOW     ; Rotate LOW right circular through C
        DEX         ; Set X to X-1
        BNE LOOP    ; Branch to LOOP if Z = 0
```

  The next lemma was the only one we needed to prove that the above code, generalized to an i-bit wide register, was correct.
- We can also prove this final example automatically. It states that rotating right an i-bit wide register through a carry flag fits back into the i-bit wide register.

$$x < 2^i \quad \wedge \quad \text{integer } i \quad \wedge \quad \text{integer } x \quad \wedge \quad (c = 0 \ \vee \ c = 1)$$
$$\Longrightarrow \text{floor}(x/2) + c \cdot 2^{i-1} < 2^i$$

Our nonlinear arithmetic extension to ACL2 provides significant benefits at a small cost. Proofs that do not involve any nonlinear inequalities are not affected and run at the same speed. A typical "small" lemma with a couple of nonlinear inequalities, which ACL2 could prove automatically before, will generally be proven within a few percentage points of the time previously required – but we can now automatically prove more of these. For more complicated lemmas and theorems, little can be said about the computer time required.

However, within broad limits, the time a user takes to complete a proof is of greater importance than the time the computer takes. Examining failed proofs and writing helper lemmas can be time-consuming and psychologically draining. The fewer lemmas a user has to prove on the way to a desired result, the better.

References

1. A. Armando and S. Ranise. A Practical Extension Mechanism for Decision Procedures. Journal of Universal Computer Science, Volume 7, Issue 2, pp. 124–140, February 2001.
2. R. Boyer and J Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. Machine Intelligence, Volume 11, pp. 83–124, 1988.
3. D. Cyrluk and D. Kapur. Reasoning about Nonlinear Inequality Constraints: A Multi-level Approach. Proceedings DARPA workshop on Image Understanding, 1989, pp. 904–915.
4. J. Harrison. Theorem Proving with the Real Numbers. Technical Report TR-408, University of Cambridge Computer Laboratory, December 1996.
5. J. Harrison. Verifying the Accuracy of Polynomial Approximations in HOL. Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'97, Springer LNCS 1275, pp. 137–152.
6. D. Kapur. A Rewrite Rule Based Framework for Combining Decision Procedures. Frontiers of Combining Systems, A. Armando, editor. Volume 2309 of Lecture Notes in Computer Science, pp. 87–102, Springer, 2002.
7. M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, 2000.
8. M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, 2000.
9. M. Kaufmann and J Moore. ACL2: An Industrial Strength Version of Nqthm. Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96), pp. 23–34, IEEE Computer Society Press, June 1996.
10. P. Miner and J. Leathrum. Verification of IEEE Compliant Subtractive Division Algorithms. Formal Methods in Computer-Aided Design (FMCAD), Volume 1166 of Lecture Notes in Computer Science, pp. 64–78.
11. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 1, Issue 2, October 1979, pp. 245–257. Springer, 1996.
12. H. Rueß and N. Shankar. Combining Shostak Theories. Proccedings of RTA 2002, LNCS 2378, pp. 1–18. Springer-Verlag, 2002.
13. D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions. LMS Journal of Computation and Mathematics, Volume 1, pp. 148–200, December, 1998.
14. R. Shostak. Deciding Combinations of Theories. Journal of the ACM (JACM), Volume 31, Issue 1, January 1984, pp. 1–12.