Integrating Nonlinear Arithmetic into 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
E-mail: hunt@cs.utexas.edu, rkrug@cs.utexas.edu, moore@cs.utexas.edu

Abstract. In this paper we present an overview of the integration of a nonlinear arithmetic reasoning package into ACL2. We provide a brief operational description of the entire arithmetic package and describe how it fits into ACL2's operation, including what was needed for the successful introduction of such a facility into an existing automated theorem prover. We describe most of the changes we made to the previous version of ACL2 as well as a couple of recent improvements to the nonlinear package we made based upon our experiences using the same. 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.

1 Introduction

It is often desirable to verify the correct operation of computer hardware or software. The use of a mechanical theorem prover such as ACL2 [7–9] provides a rigorous methodology with which to do so. The operations we wish to reason about may be arithmetic in nature; as in the floating-point hardware of a modern microprocessor, a cryptographic routine in a web browser, or pointer arithmetic in a C program. In this paper we describe our work to integrate a nonlinear arithmetic reasoning facility into ACL2, highlighting what we needed to change in ACL2 in order to use the additional reasoning power in a productive manner. We also discuss some of the heuristic details of the arithmetic package itself, and the considerations that led to those details. The arithmetic package's algorithms are described in the previous paper *Linear and Nonlinear Arithmetic in ACL2*[3].

ACL2 is the successor to the Boyer-Moore theorem prover, NQTHM. Its original linear arithmetic package's algorithms were very similar to those in NQTHM as described in *Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic*[1]. ACL2 has been under active development since 1989 and has, for example, been used for verifying the correctness of executable processor models at Rockwell-Collins, floating-point hardware at AMD, and safety-critical software in Italy. These proof efforts, among others, are presented in the ACL2 case studies volume [8].

We are working on bettering the integration of arithmetic reasoning into ACL2. This work is directed by two, complementary, principles — 1. what is obvious to the user, should be obvious to ACL2, and 2. to this end, use computer cycles, rather than human effort. Note that we have not attempted to identify a theory for which we can write a full decision procedure, nor one for which we can write an efficient semi-decision procedure. Rather, we have taken an engineering-based approach by, first, identifying a theory which a human can easily reason about but which ACL2 was particularly poor at and, second, building upon existing facilities in an incremental manner in order to strengthen ACL2's reasoning abilities. We believe that as computers get ever faster, algorithms and ideas which were previously considered too inefficient will, under appropriate limiting heuristics, become ever more practical and important.

In the next section of the paper, we present the arithmetic package's operation by example. This package may be considered as being composed of three loops. See Fig. 1. The innermost is based upon linear arithmetic. The middle one allows the user to bring to ACL2's attention some additional information about the linear properties of arbitrary functions. Facilities similar to these two loop were inherited from NQTHM and have been present in ACL2 from its inception. The outer loop is our recent contribution and handles nonlinear reasoning. Having presented the arithmetic package, we then describe how it fits into ACL2's overall operation. Section 3 thus provides the context in which the arithmetic package operates. We conclude the paper by presenting some evidence for the utility of a nonlinear reasoning facility and mentioning plans for future work.

Throughout the paper, we describe the changes made to ACL2 as well as changes to the arithmetic package. These changes were determined by engineering, rather than theoretical, principles. That is, just as many of the details of the nonlinear arithmetic algorithm were determined by experiment and observation using a carefully selected suite of challenge problems; so the desirability of most of the changes described in this paper were determined by profiling. As such, the details are affected by the structure of ACL2 itself; but we strongly believe that the overall outline and principles will apply to any (even semi-) automated theorem prover.

2 Three Quick Examples

We give here an operational view of the arithmetic reasoning package. This package can be considered to be composed of three loops[1] — linear arithmetic, partial interpretation, and nonlinear arithmetic — as shown in Fig. 1.

2.1 The Linear Arithmetic Loop

Suppose we want to prove that:

$$2 \cdot x + 7 \cdot y > 0 \quad \wedge \quad x < 1 \quad \Longrightarrow \quad 2 \cdot y \geq -1.$$

[1] For those familiar with our previous paper, we have slightly rearranged the boundaries of the arithmetic packages components as described there. In particular, `polys-from-type-set` has been moved from the linear arithmetic to the partial interpretation algorithm. The earlier arrangement more closely reflected the actual code, but the present one is pedagogically superior.

$$\begin{tikzcd}
\text{Nonlinear Arithmetic} \arrow[d] \arrow[loop right] \\
\text{Partial Interpretation} \arrow[d] \arrow[loop right] \\
\text{Linear Arithmetic} \arrow[loop right]
\end{tikzcd}$$

Fig. 1. The Arithmetic Package — A Schematic View.

We could do so by assuming the two hypotheses and the negation of the conclusion, and attempting to derive a contradiction. If one is found, the negation of the theorem is unsatisfiable (false) and so the original theorem must be true. We therefore start with the three normalized inequalities:

$$0 < y + 2/7 \cdot x \tag{1}$$
$$0 < -1 \cdot x + 1 \tag{2}$$
$$0 < -1 \cdot y + -1/2. \tag{3}$$

We now make our first pass through the loop. In a step similar to resolution, the right-hand sides of inequalities (1) and (3) will be combined by addition, yielding a new inequality which does not mention y:

$$0 < x + -7/2. \tag{4}$$

This new inequality is set aside for the next pass through the loop where it will be combined with (2), yielding

$$0 < -5/2 \tag{5}$$

This last inequality is obviously false; we have found a contradiction and therefore our proof.

In order to discuss our above actions, we need to establish some vocabulary. In the rest of the paper, the right-hand side of a normalized inequality such as those numbered above will be referred to as *polys*. Abusing our vocabulary, we will also speak of the entire inequality as a poly. Note that an equality can be normalized to two polys. The equality $x = 2 \cdot y$ is normalized to the pair $0 \leq y + -1/2 \cdot x$ and $0 \leq -1 \cdot y + 1/2 \cdot x$. A poly is said to be *about* its *heaviest* term. Terms are weighted using a lexicographic order based upon 1. the variables used in the term, 2. the function symbols used, and 3. the size of any constants used. Above, polys (1) and (3) are about y, and polys (2) and (4) are about x.

We write a poly such that the heaviest term is the left-most; we shall see some more complicated examples shortly. Note that we combined, or *cancelled*, two polys precisely when they were about the same term and this term appears with opposite sign in the two polys. Note also that the new polys, (4) and (5), were smaller than their *parents* — (1) and (3), and (2) and (4) respectively — in the sense that they are about a *lighter* term.

In each pass through this innermost loop, we one-by-one insert the new polys and cancel them with the appropriate previously inserted polys; gathering together the resulting polys for the next pass. Thus, above, in the first pass we added polys (1), (2), and (3) in that order. Upon adding (3) we canceled it with (1), and set the result, poly (4) aside for the next round. We repeat until a fixed point is reached or, as above, a contradiction is found.

We remark here that, just as for BDDs, how we weight our variables or otherwise choose the order in which to perform our cancellations — for example, above we could have combined polys (1) and (2) — does not affect termination or finding a contradiction when one exists. This procedure is complete for linear arithmetic over the rationals. (ACL2's logic includes integers and rational numbers, but no irrational numbers.)

Although the algorithm has a high worst-case run-time, overall it has demonstrated to be cheap in practice. The seemingly naive algorithm hinted at here was deliberately chosen for engineering, rather than theoretical, reasons. In particular, the following properties were all necessary:

- Incremental — Most of the time the above procedure terminates not with a contradiction, but with a stable database of polys. As we will see shortly, this database may be augmented by additional polys gathered from other sources in an attempt to allow further cancellations and, perhaps, the derivation of a contradiction.
- Non-destructive — In addition to being incrementally built up, this database is used as part of the context in which ACL2's main reasoning engine, its rewriter, operates. This rewriter is based on conditional rewrite rules whose hypotheses must be relieved before the rule can be used. If one of these hypotheses is itself an inequality, the arithmetic reasoning facilities described in this paper will be used to (attempt to) relieve that hypothesis. Whether successful or not, any work done at this time must be undone in preparation for the next rule/hypothesis.
- Quick start-up — Most of the time there are no more than a half-dozen polys around and only a couple cancellations which can be performed. Thus, any time saved by clever preprocessing or scoring potential cancellations would (probably) be swamped by the cost of doing so.

Before proceeding, we pause here to comment further on the choice to use a variant of Hodes' algorithm [4]. This choice was made based upon the results of experiments conducted during the implementation of NQTHM and should be revisited in the near future. Although, as mentioned above, the current code is usually "fast enough," better can probably be done [5]. We must, however, stress that any algorithm for deciding linear arithmetic over the rationals will have a high worst-case run-time; what matters is how an algorithm behaves given the types of problems commonly generated by ACL2 and there is little reason to believe that their distribution is random in nature.

We now describe several of the optimizations we made to this innermost loop's implementation. There are certain situations in ACL2 in which *tail-biting* can occur. From ACL2's documentation[2]: " 'Tail-biting' is our name for the insidious phenomenon that occurs when one assumes p false while trying to prove p and then, carelessly, rewrites the goal p to false on the basis of that assumption. Observe that this is sound but detrimental to success. One way to prevent tail-biting is to not assume p false while trying to prove it, but we found that too weak. The way we avoid tail biting is to keep careful track of what we're trying to prove, which literal we are working on, and what assumptions have been used to derive what results; we never use the assumption that p is false (or anything derived from it) to rewrite p to false."

The structure, a tag-tree, used to keep track of all this can grow to be quite large[3]. This created problems because every poly had one of these associated with it; and this tag-tree would have to be scanned every time the poly was cancelled against a new poly. However, within the linear arithmetic loop only a small fraction of the tag-tree was needed, so we now pull this fraction out and store it separately in a new structure, a parent-tree. Under most circumstances, this optimization has no affect on timing; but larger problems will now run to completion in a shorter time than was possible before. This will also be the effect of the other two changes described immediately below.

Previously, before adding a new poly to the database, ACL2 checked whether an equal poly had already been added. If so, the new poly was not used. However, it can be the case that ACL2 will generate sets of polys such as $0 < y+2\cdot x+5/8$, $0 < y + 2 \cdot x + 3/4$, and $0 < y + 2 \cdot x + 7/8$. Clearly, the first of these polys is stronger than the other two in the sense that it implies their truth. Thus, if the first poly is already present in the database, there is no need to add either of the other two. If conditions are such that they could be used to generate a contradiction, we would already have done so with the first.

Generalizing the above observation, we enhanced the function `poly-member` to check not whether an equal poly is already present in the database, but whether an equal or stronger poly is already present. Our notion of stronger is no more complex than that implied above. We check whether there is a poly with equal non-constant parts — the $y + 2 \cdot x$ above — and a lesser constant — the 5/8, 3/4, or 7/8 above; and if so, do not use the new, weaker, poly. Above, the first poly is stronger than the second, which in turn is stronger than the third. None are related to $0 < y + 3 \cdot x + 1$. Thus we can now recognize and filter out a greater percentage of the polys that will lead nowhere.

[2] ACL2's code is fairly readable, and is wonderfully commented. Those interested are strongly encouraged to use the source.

[3] Tag-trees are also used to report back to the user what ACL2 has done during a proof effort. This has been found to be an invaluable aid to debugging failed proofs.

We also changed the search for a proof (or contradiction) from depth-first to breadth-first. That is, if we draw a graph of the cancellation process, where each poly generated was a child of its parents, we previously explored this graph in a depth-first manner — after a cancellation, any newly created polys were recursively added to the arithmetic database before any older ones were examined.

We now collect these newly created polys into a list which we use as a push-down stack. By collecting them together, we can use `poly-member` to ensure that within any one round we only add the strongest polys. Before we push a poly onto this stack, we check whether there is a stronger poly already present, and if so discard the new one. This ensures that the stack is sorted with the strongest polys at the front and so to be seen first. Weaker polys, deeper in the stack, will be caught (as above) before being added to the database because the stronger ones were added first. Thus (within any one round) we only add the strongest polys, filtering out all the rest.

2.2 Partial Interpretation

Pure linear arithmetic is often insufficient. Suppose we wanted to prove:

$$x \in A \Longrightarrow \mathrm{size}(A) - \mathrm{size}(\mathrm{delete}(x\,,\,A)) + \mathrm{size}(B) > 0$$

There is only one poly (the negation of the conclusion) to be derived from this:

$$0 \leq \mathrm{size}(\mathrm{delete}(x\,,\,A)) + -1 \cdot \mathrm{size}(B) + -1 \cdot \mathrm{size}(A) \tag{6}$$

Clearly there is nothing linear arithmetic can do with this. But this does not mean that we have to give up. Even without reasoning about the actual definitions or details of the functions size and delete a contradiction can be derived, if we knew a couple of simple linear facts about them. More concretely, if we could use the facts that

$$\mathbf{x} \in \mathbf{A} \Longrightarrow \mathrm{size}(\mathrm{delete}(\mathbf{x}\,,\,\mathbf{A})) < \mathrm{size}(\mathbf{A}) \tag{7}$$

and

$$0 \leq \mathrm{size}(\mathbf{A}) \tag{8}$$

a contradiction can be derived. This is the purpose for which partial interpretation was designed.

All the polys about a particular term are stored together in a *pot* which is said to be *labeled* by the term which the polys are about. Thus, the single poly above is stored in a pot labeled with size(delete(x , A)). We will also loosely speak of a pot being about its label. A pot represents the conjunction of all the polys in it.

When the arithmetic database has stabilized under the linear arithmetic loop and no contradiction has been found, ACL2 will look for any newly created pots, and see if it can gather any additional polys about the new pots' labels. Here, in the first pass through the partial interpretation loop, ACL2 will look for additional information about size(delete(x , A)), and create the two new polys

$$0 < -1 \cdot \text{size}(\text{delete}(x\,,\ A)) + \text{size}(A) \tag{9}$$

$$0 \leq \text{size}(\text{delete}(x\,,\ A)) \tag{10}$$

and pass them on to linear arithmetic. Only the first of these can be cancelled with poly (6), and the result is

$$0 < -1 \cdot \text{size}(B). \tag{11}$$

In the second pass through partial interpretation the only new pot is about size(B) and so the poly

$$0 \leq \text{size}(B) \tag{12}$$

is created. This will be passed on to the linear arithmetic loop, and a contradiction will be found.

As for linear arithmetic, partial interpretation will loop until either a fixed point is reached, or a contradiction is found. At the top of each pass through this loop ACL2 will have a list of polys. These are passed off to the linear arithmetic loop where they are added to the arithmetic database. At the bottom of the loop ACL2 attempts to gather together polys about any newly created pots, as illustrated above. We do not further discuss the use or implementation of linear lemmas or polys-from-type-set here. This has already been covered in earlier literature [1, 3].

2.3 Nonlinear

Now suppose that we want to prove that:

$$3 \cdot x \cdot y + 7 \cdot a < 4 \quad \wedge \quad 3 < 2 \cdot x \quad \wedge \quad 1 < y$$
$$\Longrightarrow a < 0$$

We start with the four polys:

$$0 < -3 \cdot x \cdot y + -7 \cdot a + 4 \tag{13}$$

$$0 < 2 \cdot x + -3 \tag{14}$$

$$0 < y + -1 \tag{15}$$

$$0 \leq a \tag{16}$$

No two polys are about the same term so there are no cancellations to perform. Note however that poly (13) is about a product — $x \cdot y$ — and that polys (14) and (15) are about the product's factors — x and y. When nonlinear arithmetic is active, ACL2 will therefore multiply the left-hand sides of (14) and (15) obtaining the new poly

$$0 < 2 \cdot x \cdot y + -3 \cdot y + -2 \cdot x + 3. \tag{17}$$

The addition of this poly will allow cancellation to continue[4] and, in this case, we will prove our goal. Thus, just as ACL2 adds two polys when they have the same largest unknown of opposite signs in order to create a new smaller poly, so ACL2 can now multiply polys when the product of their largest unknowns is itself the largest unknown of another poly.

Conceptually, the nonlinear loop can be viewed as a generalization of the partial interpretation loop, and is similarly based on examining lists of newly created pots. At its simplest, if one of these new pots is about a product $(x \cdot y)$ and there are pots about its factors (x and y) ACL2 will multiply the appropriate polys together in all possible ways to generate polys about the product (all the polys about x by all the polys about y, generating a set of polys about $x \cdot y$). We do not attempt to further describe the nonlinear loop here. See [3].

We do, however, mention two heuristics we use to control the size of the search. In the Introduction to this paper we had said that we believe that as computers get ever faster, algorithms and ideas which were previously considered too inefficient will, under appropriate limiting heuristics, become ever more practical and important. The complexity of the algorithm hinted at above is (at least) exponential. We do not mean to say that faster computers will alone ever be able to overcome such obstacles. However, our goal in integrating this algorithm into ACL2 was not to implement a complete decision procedure[5]. Rather, we wished to render obvious to ACL2 what is obvious to the user. We therefore attempted to achieve a balance between reasoning power and resource consumption. Experimentation led us to the following two limiting heuristics. These have both proved vital to making the algorithm of practical use. First, we limit the number of passes through the nonlinear loop to three. Second, when nonlinear arithmetic is active and the list of polys waiting to be added to the database grows too large, we prune this list by keeping only the shortest polys — those with the least number of addends. These heuristic limitations seem to be weak enough to allow ACL2 to catch a sufficient number of nonlinear facts, but strong enough to limit the time spent in a proof effort to a not unreasonable amount. They should probably be revisited in a few years when computers are even more powerful than they are today.

In order to support nonlinear arithmetic, we changed the partial interpretation loop. When nonlinear arithmetic is disabled, the partial interpretation loop behaves as described in the previous section. This is the original behavior. However, when nonlinear arithmetic is active, ACL2 is a little more aggressive in collecting information. When a new pot is about a product, say $\text{size}(B) \cdot \text{size}(A) \cdot x$, ACL2 will attempt to gather information about each of the terms in the *exploded* pot label; in this case each of the terms $\text{size}(B)$, $\text{size}(A)$, x, and $\text{size}(B) \cdot \text{size}(A) \cdot x$.

[4] Poly (17) can be canceled with (13). The result can be canceled with (15), and so on. The final cancellation will be with the negation of our goal

[5] We couldn't, even if we wanted to. ACL2's logic does not include the irrational numbers, and it has been proven that there is no complete algorithm for nonlinear arithmetic over the rationals [6].

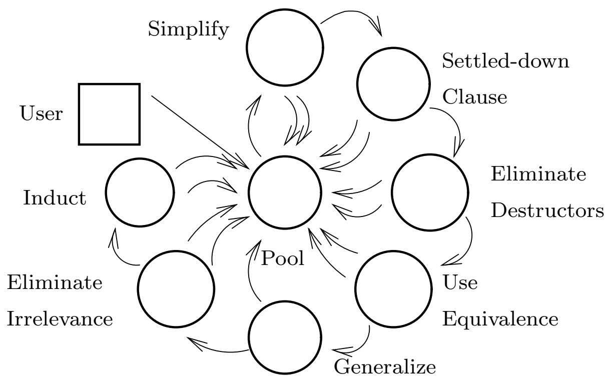

Fig. 2. ACL2 — a high level view.

ACL2 thereby has a chance to gather polys about a products factors in preparation for nonlinear arithmetic[6].

We conclude this section with a couple of examples of incompleteness in our algorithm:

– It is a theorem in ACL2 that $\forall x \,.\, x \cdot x \neq 2$ [2]. There is nothing that our algorithm can do with this.
– Similarly, since there are no cancellations to perform nor factors to multiply, there is nothing our algorithm can do with $0 < a \cdot b \ \wedge\ 0 < c \cdot d \ \wedge\ 0 < a \cdot c \implies 0 < b \cdot d$. (However, in conjunction with our library of arithmetic rules, this is automaticly proven)

3 ACL2 — an Overview

We have described the arithmetic package. We now describe the architecture of ACL2 and how the arithmetic package fits into this.

At a high level, ACL2 is organized as follows. At the center is a pool of goals to be proved; see Fig. 2. Initially, the user's submitted theorem is the only goal in this pool. Surrounding the pool are seven proof procedures — Simplify, Settled-down Clause, Eliminate Destructors, Use Equivalence, Generalize, Eliminate Irrelevance, and Induct. Each goal is successively withdrawn from the pool and given to the first proof procedure, Preprocess. Each procedure either *hits*, reducing the goal to n further (presumably simpler) goals and returns these to the pool; or *misses*, and passes the goal on to the next procedure in line. If n is 0, the goal has been proved by the procedure; when the pool is empty, the original

[6] Consider $x \in A \ \wedge\ y \in B \implies \text{size}(\text{delete}(x\,,\,A)) \cdot \text{size}(\text{delete}(y\,,\,B)) > \text{size}(A) \cdot \text{size}(B)$. This is a good example to work on one's own.

theorem has been proved. If all the procedures miss (or the user interrupts the proof process), the proof attempt has failed[7].

As of the forthcoming ACL2 version 2.8, the activation of nonlinear arithmetic will change. Previously, when the user enabled nonlinear arithmetic it was active all the time. Under the new behavior, nonlinear arithmetic will not be active until the goal has stabilized under simplification (rewriting). Before this change was made, when nonlinear arithmetic was enabled ACL2 could easily get swamped at certain points in a proof — most notably, at the very beginning and just after destructor elimination or generalization. This is no longer a problem. Thus, unlike some of the other changes described in this paper which make larger problems more practical, this change has resulted in a general speedup of ACL2 when nonlinear arithmetic is enabled even for smaller problems.

We now briefly describe the implementation of this change. There is a flag, `stable-under-simplificationp`, which can be considered to be true if and only if the goal currently being examined had been given to the simplifier and subsequently passed on to Settled-down Clause unchanged. Toggling this flag is then counted as a hit for this (rather simple) proof procedure[8]. This flag was originally introduced in order to allow *staged* simplification of large microprocessor models. By using computed-hints and this flag, we can prevent the microprocessor's state transition function from expanding the symbolic representation of a state until that state has been maximally simplified. By using these same facilities to turn on the use of nonlinear arithmetic, the easy work is accomplished cheaply just as before nonlinear arithmetic was introduced; however, when this has achieved as much as possible we then activate nonlinear arithmetic to (hopefully) finish the job.

In this paper, our primary focus is the Simplifier. It can be divided into two major pieces — the Lemma Library (ACL2's evolving knowledge base), and the Simplifier proper. See Fig. 3. The Simplifier proper can be further subdivided into several component reasoning packages. There are three major packages — a rewriter, a type-reasoning package[9], and the arithmetic package. There is also a tautology checker as well as a few other minor pieces which we will lump together under the rubric "other".

The rewriter, the arithmetic package, and the type-reasoning package are all affected by the state of the lemma library as indicated by the dashed arrows. This library contains the rules/lemmas which extend and guide the operations

[7] Once a proof attempt has been started, the only interaction the user can have to guide the proof is to interrupt it. In this sense, ACL2 is entirely automatic with no user guidance possible. A user typically guides a proof in two ways — 1. A judicious selection of rewrite rules or lemmas to enhance ACL2's "knowledge base"; and 2. The use of hints such as :case-split, :use, :do-not, or :induct. Hints must be attached to the theorem at the time of its submission to ACL2.

[8] Settled-down Clause is in charge of adjusting several other heuristic flags and settings, primarily related to induction, not described here.

[9] Although we do not describe the type-reasoning package here, we do wish to mention that this package does not perform type-reasoning as it is commonly understood in the functional programming community.

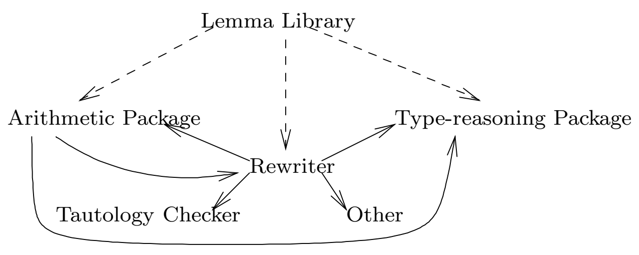

Fig. 3. The Simplifier.

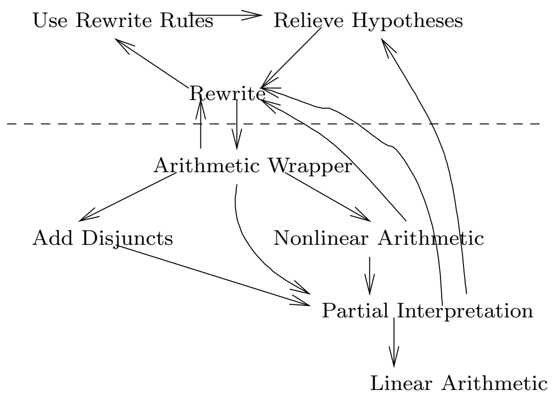

Fig. 4. Arithmetic and the Rewriter.

of these three packages. It is through the judicious use of these rules that the user can guide ACL2's operations. For the purposes of this paper, only two types of rules are relevant — rewrite rules which guide (are used by) the rewriter, and linear lemmas which guide the partial interpretation loop.

The solid arrows in Fig. 3 indicate which packages can call another package. Note that the rewriter and the arithmetic package are mutually recursive. In this paper we are concerned with the arithmetic package and its relationship to the rewriter[10].

[10] We remark here that the arithmetic package has since the beginning been able to call the type-reasoning package. Although not further discussed here, we have recently added the ability of the type-reasoning package to call the linear arithmetic part of the arithmetic package. This latter fact is not represented in the figure.

We now describe the relationship between the Rewriter and the Arithmetic Package. See Fig. 4. We first discuss the rewriter and arithmetic package separately, and then the arrows linking them.

We divide the rewriter into three parts — Use Rewrite Rules, Relieve Hypotheses, and Rewrite which should be considered to be the main entry point or wrapper function.[11] These three pieces form a mutually recursive clique of functions as indicated in the figure.

ACL2's rewriter is based upon the use of conditional rewrite rules. Given a term to rewrite, ACL2 will find all the applicable rewrite rules for the term and successively try to apply these rules until one succeeds. A rule succeeds when all of its hypotheses have been relieved. A hypothesis is *relieved* by instantiating any free variables and rewriting it to true. Thus, the mutual recursion in the rewriter. Note here that if a hypothesis is an inequality or an arithmetic equality, the rewriter hands it over to the arithmetic package to decide. This will be touched upon again shortly.

We divide the arithmetic package into five parts — the three loops already presented, a main entry point/wrapper function labeled Arithmetic Wrapper, and a piece labeled Add Disjuncts. This last piece is concerned with negated equalities. Just as an equality is equivalent to the conjunction of two polys, so a negated equality is equivalent to the disjunction of two polys — e.g., $x \neq y$ is equivalent to $x < y \ \vee \ y < x$. The arithmetic package finds a proof by looking for a contradiction. Thus, if each of the two disjuncts lead to a contradiction when added individually to the arithmetic database, the original negated equality leads to a contradiction and we have found our proof. If neither disjunct leads to a contradiction, there is nothing we can do and we return our original arithmetic database unchanged. (Recall that the arithmetic database represents the conjunction of the polys in it.) If only one disjunct leads to a contradiction, we can return the augmented arithmetic database to which the other disjunct was added. Add Disjuncts handles the general situation in which there may be more than one of these negated equalities. Clearly, this can suffer from exponential blowup.[12] We therefore prevent the use of nonlinear arithmetic in this situation.

The arithmetic package's entry point normalizes the inequalities and equalities passed in by the rewriter, and passes the resultant polys off to the appropriate place. All of the polys stemming from inequalities and (un-negated) equalities are handed off first — to the nonlinear arithmetic or partial interpretation loops as determined by whether nonlinear arithmetic is currently being used. It is only after the database has stabilized under these operations that the polys resulting from negated equalities are handed off to add disjuncts.

All that is left to describe now are the calls from the arithmetic package to the rewriter. There are five of these as shown in Fig. 4. We first describe the call from Rewrite to the Arithmetic Wrapper. When ACL2 is rewriting a term

[11] ACL2's rewriter is rather complex. We ignore almost all of this complexity here, and instead present only what is needful for this paper.

[12] There is one theorem in ACL2's regression suite which leads to a goal in which there are sixty-four negated equalities.

it keeps track of the objective for that term — approximately, what it wants to rewrite that term to. This objective can be any one of three values; “true”, “false”, or “unknown.” In general the objective is “unknown”, but there are a few situations in which it is known. The simplest of these is that a hypothesis of a rule needs to be rewritten to true for the rule to be relieved. When ACL2 is rewriting a term which is an inequality or an arithmetic equality and the objective is either “true” or “false”, it will pass the (posssibly negated) term off to the arithmetic package.

We now move on to discuss the calls from the arithmetic package to the rewiter. Recall that the partial interpretation loop is in charge of using linear lemmas to generate additional polys. These lemmas may include hypotheses which need to be relieved. This accounts for the arrow from Partial Interpretation to Relieve Hypotheses. Note that if one of a linear lemma’s hypotheses is an inequality, then it will be passed back to the arithmetic package by the rewriter. We see here another example of why the algorithms used in the arithmetic package must be incremental and non-destructive.

The conclusions of these linear lemmas are equalities and inequalities. The right- and left-hand sides of these conclusions are rewritten before being normalized and passed on to the linear arithmetic loop. This is represented by the arrow from Partial Interpretation to Rewrite. We comment here that this is done to enforce the use of any desired normal forms. A somewhat contrived example is the following. In lisp, and therefore ACL2, `floor` is a binary function which returns the greatest integer less than or equal to the ratio of its arguments. That is, we have the relationship:

$$\text{floor}(a,\ b) <= a/b. \tag{18}$$

Let us assume that we have encoded this as a linear-lemma and the partial interpretation loop has just encountered a newly created pot about $\text{floor}(c{\cdot}x,\ x)$. Let us also assume that $x \neq 0$. Without rewriting, we would add the poly

$$0 \leq -1 \cdot \text{floor}(c \cdot x,\ x) + c \cdot x/x. \tag{19}$$

This is not nearly as desirable as

$$0 \leq -1 \cdot \text{floor}(c \cdot x,\ x) + c. \tag{20}$$

Moving on, the multiplication of polys done by the nonlinear arithmetic loop is accomplished by passing the problem off to the rewriter, and letting it simplify the product. This is the arrow from nonlinear arithmetic to the rewriter. It is also the source of an obvious potential optimization. One could write some special-purpose code which did the job much more efficiently. We have not yet done so because the increased flexibility of the current method has greatly aided our experiments. When the nonlinear arithmetic algorithms and their integration with ACL2 has stabilized, we will almost certainly take advantage of this optimization.

However, the situation mentioned above is not as dire as it might seem. Previously, the calls to the rewriter from within the arithmetic package would use whatever (often large) set of rewrite rules was being used for general rewriting. We now allow the use of a specially constructed minimal theory for this rewriting, when nonlinear arithmetic is active. This allows the rewriting to proceed more quickly. It also allows us to use a different normal form within the arithmetic package than outside it. This has proven very useful — for instance, nonlinear arithmetic can reason about $x^y \cdot x^z$ better than it can about the equivalent x^{y+z}. This latter form, however, is often better for the rewriter. With the introduction of this new minimal theory, we needed to allow the arithmetic package to rewrite equalities and inequalities before normalizing them into polys. Thus the final arrow, from the Arithmetic Wrapper to Rewrite.

4 Conclusion

The combination of the new nonlinear package and improved arithmetic libraries has allowed us to prove more theorems more automatically. Initial feedback from early users has been generally positive. In particular, users felt that the amount of thought and effort that they had to put into completing a proof involving arithmetic was often reduced. We continue to improve the heuristics and further integrate the arithmetic libraries with these new facilities.

A couple of pieces of evidence of the increased power now available are:

– In conjunction with our library of arithmetic lemmas, we can automatically prove that rotating an i-bit wide register through a carry flag fits back into the i-bit wide register:

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

– Experiments showed that more than one-half of the helper lemmas — those used merely to prove one of the desired theorems, but not otherwise useful — and three-quarters of the (nontrivial) hints in the IHS books were no longer needed. IHS stands for Integer Hardware Specification. These books are a collection of theorems and rules designed to facilitate reasoning about bit-level machine operations as if they were arithmetic operations on integers. These books involve much use of arithmetic and have been used as a source of challenge problems during our work.

References

1. 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.

2. R. Gamboa. Square Roots in ACL2: A Study in Sonata Form. UTCS Tech Report TR96-34, November, 1996.
3. W. Hunt, Jr., R. Krug, and J Moore. Linear and Nonlinear Arithmetic in ACL2. CHARME 2003.
4. L. Hodes. Solving Problems by Formula Manipulation in Logic and Linear Inequalities. Proceedings of the Second International Conference on Aritficial Intelligence, pp. 553-59, 1971.
5. P. Janičić, I. Green, A. Bundy. A Comparison of Decision Procedures in Presburger Arithmetic. Research Paper #872, Division of Informatics, Univ. of Edinburgh, October 97.
6. J. Robinson. Definability and Decision Problems in Arithmetic. Journal of Symbolic Logic, Volume 14(2), pp. 98-114, 1949.
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.