Proving A Specific Type of Inequality Theorems in ACL2

A bind-free experience report

Hanbing Liu
Advanced Micro Devices, Inc.
7171 Southwest Parkway
Austin, Texas, 78735
hanbing.liu@amd.com

ABSTRACT

We describe how we guide ACL2 to follow a divide-and-conquer strategy for proving inequalities of the type $|P(\vec{e})| \leq C$. $P(\vec{e})$ is a polynomial in variables $\vec{e}$ and C is a constant.

Our approach involves (1) writing an ACL2 program to estimate the upper-bound of such polynomials and (2) using the bind-free mechanism to integrate the upper-bound estimation program to guide rewriting. We think it is interesting to showcase how we extract the relevant information from the hypothesis and how such information is used to influence rewriting.

Techniques like ours can be useful to ACL2 users who want to better control rewriting when their problems share specific characteristics with our $|P(\vec{e})| \leq C$ type problem.

Categories and Subject Descriptors

G.1.0 [**Numerical Analysis**]: General—*error analysis*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*; F.4.1 [**Mathematical Logic And Formal Languages**]: Mathematical Logic—*mechanical theorem proving*

General Terms

Algorithm, error analysis, verification, theorem

Keywords

ACL2, bind-free, rewriting strategy, ACL2 free variables

1. PROBLEM

Let $P(\vec{e})$ be a polynomial in variables $\{e_i\}$. Suppose we know that the $|e_i| \leq c_i$. We want to show that some constant C is the upper bound of the polynomial by proving a corresponding ACL2 theorem.

If $P(\vec{e})$ is simple, one may rely on ACL2's linear reasoning together with its arithmetic library to prove directly that $|P(\vec{e})| \leq C$. For example, equipped with `arithmetic-4` library, the ACL2 theorem prover can prove the following theorem `p2` easily, where the polynomial is simply $e_1 + e_2$, while $c_1 = c_2 = 1$.

```
(include-book "arithmetic-4/top" :dir :system)
(defthm p2
   (implies (and (<= (abs e1) 1)
                 (<= (abs e2) 1))
            (<= (abs (+ e1 e2)) 2)))
```

The ACL2 rewriter and its built-in linear reasoning are no longer effective when $P(\vec{e})$ is complex. Proving `p10`, a larger variation of `p2`, is more time consuming. The `p10` theorem is only proved after ACL2 explores tens of thousands of cases.

```
(defthm p10
   (implies (and (<= (abs e1) 1)
                 (<= (abs e2) 1)
                 (<= (abs e3) 1)
                 ...
                 (<= (abs e9) 1)
                 (<= (abs e10) 1))
            (<= (abs (+ e1 e2 e3 ... e9 e10)) 10)))
```

If given a `p100` theorem to prove, the ACL2 theorem prover is likely to either run out of memory or take a very long time if it were to follow the same strategy that it used to prove the `p10` theorem.

When $P(\vec{e})$ is complex, the ACL2 built-in linear procedures and strategies (as embodied in the its arithmetic library) are too general to be effective.

We need a more focused strategy to prove such inequalities.

2. DIVIDE-AND-CONQUER STRATEGY

One may attempt to guide ACL2 by first proving the following rules directly. A new ACL2 user may even hope that the ACL2 theorem prover can use these rules to prove the `p100` theorem automatically.

```
   abs(term) <= d1        |    abs(fact) <= d1
   abs(poly) <= d2        |    abs(term) <= d2
   d1+d2 <= C             |    d1*d2 <= C
=>                        |  =>
   abs(term + poly) <= C  |    abs(fact * term) <= C
```

To use the first rule, ACL2 first needs to pick suitable bindings for d_1 and d_2. Next, ACL2 needs to *back-chain deeply* — attempting to first prove that $|term| \leq d_1$, $|poly| \leq d_2$, and $d_1 + d_2 \leq C$, before it can conclude $|term + poly| \leq C$.

Unfortunately, in this case, both d_1 and d_2 are considered "free variables" by the ACL2 theorem prover because neither d_1 nor d_2 appears in the left-hand side of the rewrite rule. When the ACL2 theorem prover attempts to use the rule to rewrite some term, it "one-way" unifies the term against the left-hand side of the rewrite rule: `abs(term + poly) <= C`[1]. When the unification is successful, ACL2 finds suitable substitutions for `term`, `poly`, and `C`. However, because d_1 and d_2 do not appear in the left-hand side of the rewrite rule, the unification does not provide any useful information for how to pick suitable substitutions for d_1 and d_2.

We also want to point out that, in order to prove $|poly| \leq d_2$, ACL2 needs to follow the same procedure again and again, thus recurring deeply until the subgoal $|poly_i| \leq d_{2_i}$ is simple enough for it to recognizee. Thus, to follow this focused strategy successfully, ACL2 needs not only to pick an initial feasible split of C into d_1 and d_2, but also to pick a sequence of feasible splits, one for each $|poly| < d_2$ subgoal that would arise during the recursion.

To nudge the ACL2 theorem prover into following this very focused back-chain strategy, we now have two separate tasks. The first task is to define an algorithm that can pick out suitable d_1 and d_2 intelligently. The second task is to integrate such an algorithm to control how the ACL2 theorem prover does its back-chain reasoning.

In this paper, we present one such simplistic algorithm for picking out suitable ds for a simple class of $|P(\vec{e})| \leq C$ problems. We would like to note that how to design an powerful algorithm is not our focus here. Our focus is on how one may integrate such an algorithm to control rewriting.

3. ALGORITHM

We need an algorithm that can pick out suitable d_1 and d_2 so that $|term| \leq d_1$ and $|poly| \leq d_2$ are provable. As a minimum requirement, we need the algorithm to pick out some d_1 and d_2 such that $|term| \leq d_1$ and $|poly| \leq d_2$ are true.

An upper-bound estimation algorithm fits the purpose. Such an algorithm takes two inputs, a description of the polynomial and a description of the constraints, $\{|e_i| \leq c_i\}$. It returns an upper bound for the polynomial.

A simplistic algorithm may just go over the expression that represents the polynomial. It recursively calls itself to get an upper bound on the sub-expressions and combines the upper bounds on the sub-expressions in a conservative way to compute an upper bound of the original expression.

We have implemented such an algorithm.

```
(defun upper-bound-c-or-var (c_or_var bindings)
  (if (acl2-numberp c_or_var)
      (abs c_or_var)
    (if (symbolp c_or_var)
        (cdr (assoc-equal c_or_var bindings))
      (hard-error 'upper-bound-c-or-var
                  "Syntax error: ..."))))

(defun upper-bound-expr (expr bindings)
  (cond
   ((not (consp expr))           ;; (1) var
    (upper-bound-c-or-var expr bindings))

   ((equal (car expr) 'quote)    ;; (2) const
    (upper-bound-c-or-var (cadr expr) bindings))

   ((equal (car expr) 'binary-*) ;; (3) (* x y)
    (* (upper-bound-expr (cadr expr) bindings)
       (upper-bound-expr (caddr expr) bindings)))

   ((equal (car expr) 'binary-+) ;; (4) (+ x y)
    (+ (upper-bound-expr (cadr expr) bindings)
       (upper-bound-expr (caddr expr) bindings)))

   ((equal (car expr) 'unary--)  ;; (5) (- x)
    (abs (upper-bound-expr (cadr expr) bindings)))

   (t (hard-error 'UPPER-BOUND-EXPR
                  "Syntax error: ~p0~%"
                   expr))))
```

The upper bound found by a simplistic algorithm like ours may not be sufficient to establish some difficult $|P(\vec{e})| \leq C$ results, but for typical problems that we encountered, this simplistic algorithm works[2].

4. INTEGRATION WITH ACL2

When our upper-bound estimation program returns a number that is no greater than C, we know that $|P(\vec{e})| \leq C$ is true. However, we prefer to have an ACL2 theorem stating that $|P(\vec{e})| \leq C$. This is because we often want to use such a result as a proven lemma to prove some other results.

Unfortunately, getting a positive answer to a $|P(\vec{e})| \leq C$ question, via a program run, does not give us an ACL2 theorem about the polynomial itself. We integrate the upper bound estimation algorithm as a heuristic to guide ACL2 to prove such a theorem.

[1]To be precise, the real left-hand side is `abs(term + poly) > C`. The ACL2 theorem prover is trying to rewrite it to `nil`.

[2]In our work, $P(\vec{e})$ typically represents the relative error in the output of some approximation algorithm. These algorithms are *self-correcting*, in that, rounding errors introduced in the early iterations will dissipate quickly in later iterations. This must have something to do with why our simplistic algorithm works.

Our integration technique is simple. We first define a function `bind-d1-with-hints`.

```
(defun bind-d1-with-hints (expr hints)
  (list (cons 'd1
        (list 'quote
              (upper-bound-expr expr hints)))))
```

The function returns a `'((d1 . <upper bound>))`, where `<upper bound>` is a constant returned by `upper-bound-expr` on ACL2's representation of the polynomial `expr`. We expect that the constraints are encoded in the `hints` variable as a list of pairs of form `(varname . c)`.

We then prove a few carefully chosen rewrite rules. There are only five of them. Each corresponds to a branch in the `upper-bound-expr` algorithm. Two of them are listed here[3].

```
(defthm over-estimate-rule-var-leaf
  (implies
   (and (syntaxp (symbolp x))
        (bind-free (bind-d1-with-hints x hints) (d1))
        (less_equal_than (abs x) d1)
        (<= d1 C))
   (less_equal_than_with_hints (abs x) C hints)))
```

The rewrite rule says: if we are trying to prove $|x| \leq C$, and we know that x is a `symbolp`, we will first compute its upper bound d_1 using `bind-d1-with-hints`, and then try to prove $|x| \leq d_1$ and $d_1 \leq C$.

```
(defthm over-estimate-rule-add
  (implies
   (and (bind-free (bind-d1-with-hints x hints) (d1))
        (less_equal_than_with_hints (abs x) d1 hints)
        (less_equal_than_with_hints (abs y)
                                    (+ (- d1) C)
                                    hints))
   (less_equal_than_with_hints (abs (+ x y))
                               C hints)))
```

This rewrite rule says: to prove an inequality of form $|x + y| \leq C$, we first compute the upper bound d_1 for the absolute value of sub-expression x; we will then attempt to prove that $|x| \leq d_1$ and $|y| \leq (C - d_1)$.

One might be tempted to think that only the subgoal $|y| \leq (C - d_1)$ is the *interesting* while the $|x| \leq d_1$ subgoal is trivial to prove. We would like to emphasize that this is not the case. To prove $|x| \leq d_1$, the ACL2 theorem prover may need to invoke `bind-d1-with-hints` again to decide on how to split the constant d_1 for creating new subgoals and continue the backchain reasoning. In short, the process of proving subgoal $|x| \leq d_1$ is very much like the process of proving $|y| \leq (C - d_1)$. Furthermore, both processes are very similar to the process of proving $|x + y| \leq C$.

[3]One may be more interested in checking out this paper's supporting material in the ACL2 2009 workshop archive.

The function `less_equal_than_with_hints` mentioned in the previous rules is defined as

```
(defund less_equal_than_with_hints (x d hints)
  (declare (ignore hints))
  (<= x d))
```

A typical $|P(\vec{e})| \leq C$ theorem that we prove is:

```
(defthm q0u-relative-error-bounded
  (implies
    (and (rationalp a)
         (rationalp b)
         (rationalp e)
         (not (equal b 0))
         (not (equal a 0))
         (less_equal_than (abs e) (expt 2 -14))
         (rationalp rne2)
         (less_equal_than (abs rne2) (expt 2 -64))
         (rationalp rne3)
         (less_equal_than (abs rne3) (expt 2 -64))
         (rationalp rne3)
         (less_equal_than (abs rne4) (expt 2 -64)))
   (less_equal_than_with_hints
     (relative-err
       (mylet* ((y0 (y0 b e))
                (e0 (e0 b y0 rne2))
                (y1 (y1 y0 e0 rne3))
                (y2 (y2 y0 y1 e0 rne4))
                (q0u (q0u a y2)))
               q0u)
       (* a (/ b)))                ;; |P(e_bar)|
     (expt 2 -41)                  ;;  C
     '((e . 1/16384)               ;;  hints
       (rne2 . 1/18446744073709551616)
       (rne3 . 1/18446744073709551616)
       (rne4 . 1/18446744073709551616)))))
```

The theorem says the relative error between $q0u$ and $\frac{a}{b}$ is smaller than 2^{-41}, where $q0u$ is computed via a five-step algorithm (as encoded in the `mylet*` term)[4]. $Rne2$, $rne3$, and $rne4$ represent rounding errors introduced in the corresponding steps. The `mylet*` is a macro that we defined. It eagerly expands a term by replacing occurrences of variables with their bindings.

There are several interesting aspects of this theorem. First, the ACL2 theorem prover proves this theorem automatically after we proved our five rewrite rules. We note that the relative error term in the left hand side of the conclusion expands into a 7-degree polynomial of four variables with 27 terms. Many of the terms have more than three multiplicative factors. Without our technique of invoking an upper bound estimation algorithm to find bindings for free variables in our rewrite rules, the plain ACL2 will not be able to prove this inequality.

[4]This is Algorithm 8.8 from the text book *IA-64 and elementary functions: speed and precision* by Peter Markstein [2]. $Q0u$ is an unrounded approximation of $\frac{a}{b}$. The later steps round the value to produce a single-precision IEEE result.

Second, let us look at the left-hand side of our conclusion. We are able to write $relerr(v, a/b) \leq 2^{-41}$ in the theorem instead of writing |the expanded form| $\leq 2^{-41}$. Lemma such as `over-estimate-rule-add` is attempted automatically after ACL2 has expanded the non-recursive definitions of `relative-err`, `y0`, `e0`, `y1`, `y2`, and `q0u` and has normalized the expression into the form `(less_equal_than_with_hints (abs ...) ...)`.

Third, we also note how the `hints` are supplied. We see that `(less_equal_than e (expt 2 -14))` is in the hypothesis. We know that $|e| \leq 2^{-14}$. We extract this assumption and encode it as a pair in the hints, `(e . 1/16384)`. We check that $1/16384 = 2^{-14}$. This ensures that the description of the constraints (i.e., `hints`) matches the actual constraints (i.e., the theorem's hypothesis). This ensures that the value returned by `upper-bound-expr` is, in fact, an upper bound of the polynomial.

5. CONCLUSION

A $|P(\vec{e})| \leq C$ type problem has the following characteristics:

- $P(\vec{e})$ has an internal structure. It can be constructed from sub-components via a small set of operators. In the case of polynomials, such operators include `binary-+`, `binary-*`, and `unary-`. Each sub-component is similar to $P(\vec{e})$ in that they have similar internal structure.
- C, on the other hand, has no apparent internal structure. It is not obvious how to pick out a suitable split of C that matches the natural decomposition of $P(\vec{e})$, even if we know that such a suitable split exists.

As a result, there is no obvious "divide-and-conquer" strategy for the ACL2 theorem prover to follow in proving results of this type.

In this paper, we described a way for ACL2 to find and follow such a "divide-and-conquer" strategy. Instead of supplying the many explicit hints on how to split C to match the decomposition of $P(\vec{e})$, we use an algorithmic method to compute the suitable split of C by examining the decomposition of $P(\vec{e})$. We provide the split of C as bindings for free variables in a set of rewrite rules. It is done via the ACL2 *bind-free* mechanism.

The approach is both simple and effective. It works well with ACL2's existing arithmetic library.

6. FUTURE WORK

We want to note that the approach is also extensible.

For example, one may allow the `hints` variable to hold a user-defined data structure instead of just a list of pairs. One can design a different upper bound estimation algorithm to use such a data structure. One may even introduce suitable rewrite rules that update the data structure as the rewriting and back-chaining are in progress.

We think that an approach similar to ours can be used to prove other types of theorems as long as they share the characteristics that we described earlier.

7. AFTERWORD

At Advanced Micro Devices,Inc., we use ACL2 to verify floating point arithmetic designs. A part of this work is to verify that certain iterative div/sqrt algorithms are correct. To prove such an algorithm correct, we often need to show that the relative error between some approximation and the true value is smaller than some threshold after a fixed number of iterations.

We considered several meta-level approaches in guiding ACL2 to prove this type of inequality[5].

Our first thought is to prove a "metatheorem."

However, it is difficult to come up with a suitable metafunction and/or hypothesis metafunction that we can prove to be correct. Furthermore, for each new improvement of the upper-bound estimation algorithm, we need to prove a new metatheorem again.

We also note that in our current approach, we don't even need our upper-bound estimation algorithm to be correct. We only need that the specific runs of the algorithm return correct upper bounds. If a specific run of the algorithm returns a wrong result, the conjecture will not be proved. If we were to prove the metatheorem, we would have to define a complicated hypothesis metafunction to identify and reject all cases in which the upper-bound estimation algorithm may return a wrong result.

Another idea is to use the *computed-hints* facilities in ACL2. We are not familiar with writing computed hints. We are concerned that, because the $P(\vec{e})$ is complex, we may need to insert too many `:use` hints to be useful to ACL2. We also need to decide to when to insert these hints, possibly by controlling the case split using `:cases` hints.

In the end, we think our approach of using the upper-bound estimation in a `bind-free` hypothesis is both simple and effective. The trick of using a "dummy" variable `hints` in the conclusion of the conjecture to supply context information worked out nicely. This avoided the complexity of examining the actual ACL2 state to collect constraint information.

Acknowledgments

Sincere thanks for the encouragement and support from my colleagues at Advanced Micro Devices, Inc.. Many thanks to our reviewers for their helpful advices and recommendations.

8. REFERENCES

[1] W.A. Hunt Jr., M. Kaufmann, R.B. Krug, J Moore, and E.W. Smith. Meta reasoning in ACL2. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, number 3603 in Springer Lecture Notes in Computer Science, pp. 163–178, 2005.

[2] P. Markstein. *IA-64 and elementary functions: speed and precision*. Hewlett-Packard professional books. Prentice Hall PTR, 2000.

[5]For a discussion of available meta-level reasoning mechanism in ACL2, see [1].