Using Testing to Automate Proofs

John Erickson
Intel Corporation
john.erickson@intel.com

Overview

- Created a clause processor that prunes subgoals prior to induction
- Clause is a disjunction of terms, so throwing out terms is sound
- Recent coherence protocol almost entirely automated using this technique
- General technique could be adopted to any other proof

I'd like to talk about a recent proof I did of a coherence protocol that was almost entirely automated after using a clause processor that prunes subgoals before induction. This technique is completely general and could be used to greatly increase the automation of any proof in ACL2.

Why Prune?

- Subgoals can contain many irrelevant terms
- Sorting through which are important manually is taxing
- Random testing can quickly reveal which terms can be abandoned (with high probability)
- If pruning misgeneralizes a theorem, no harm done (proof fails)
- Proofs fail quickly when using testing

So Why Prune?

The main reason is that

Subgoals can contain many irrelevant terms

And that

Sorting through which are important manually is taxing

In the next few slides I will give some examples of how pruning can be helpful. I will also go into some of the details of how we prune clauses efficiently.

Luckily,

Testing can quickly reveal which terms can be abandoned (with high probability)

And

If pruning misgeneralizes a theorem, no harm done (proof fails)

One thing to point out here is that although testing is highly effective, it isn't perfect.

But if it prunes some terms that it should not, it does not introduce unsoundness. It just create a new subgoal that cannot be proven. Usually this is discovered quickly because it will lead to another subgoal that is easily refuted. At that point, the clause processor will return the empty clause which causes ACL2 to abandon the proof attempt.

The underlying assumption here is that you have some kind of effective testbench where you can use small models to test theorems such that the results can be carried over to the more general theorems you are trying to prove. I think that effective testbenches can be constructed for most types of theorems. In fact, for many theorems they can probably be constructed automatically using some kind of random testing. The ACL2 Sedan has this kind of capability built in and I have found it to be highly effective. Unfortunately, for this work, I was unable to leverage that infrastructure because I needed a little more control over how testing was used. But I can imagine a general facility that combines these ideas with what has been implemented there.

Pruning Example

```
((NOT (RSPP ARP))
 (NOT (CCHSTATEP ACE))
 (NOT (INTEGERP ATG))
 (< ATG '0)
 (NOT (CACHESP SCS))
 (NOT (GOOD-STATE (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                       SCS)))
 (NOT (GOOD-STATE-M (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                         SCS)))
 (NOT (MYSTEP-GOOD (EVENT ETD 'REQE ESD 'DIR)
                   (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                        SCS)))
 (NOT (INTEGERP ETD))
 (< ETD '0)
 (NOT (INTEGERP ESD))
 (< ESD '0)
 (NOT (LOOKUP ETD SCS))
 (NOT (LOOKUP ESD SCS))
 (NOT (LOOKUP I SCS))
 (NOT (INTEGERP I))
 (< I '0)
 (NOT (EQUAL (AGENT-TXN (LOOKUP I SCS))
             'REQE))
 (EQUAL (AGENT-RSP (LOOKUP I SCS)) 'GNTE)
 (NOT (EQUAL (AGENT-TXN (LOOKUP ESD SCS))
             'REQE))
 (AGENT-TAG (LOOKUP ESD SCS))
 (< (DIST-TO-HEAD
      I
      (SYS (AGENT 'NIL ARP 'REQE ACE ATG)
           (UPDATE ESD
                   (AGENT (AGENT-REQ (LOOKUP ESD (CLEAR-ALL SCS)))
                          'GNTE
                          (AGENT-TXN (LOOKUP ESD (CLEAR-ALL SCS)))
                          (AGENT-CCHSTATE (LOOKUP ESD (CLEAR-ALL SCS)))
                          (AGENT-TAG (LOOKUP ESD (CLEAR-ALL SCS))))
                   (CLEAR-ALL SCS))))
    (DIST-TO-HEAD I
                  (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                       SCS)))
 (EQUAL
    (DIST-TO-HEAD
      I
```

Don't read this

So just to give a concrete idea of what I'm talking about, imagine you have a theorem which you are trying to prove by induction. It splits into several hundred subgoals, each of which has 30 terms or so in it. These fail to prove, so you end up looking at something like this. Wouldn't it be nice if someone could just tell you what the interesting part of this subgoal was?

Pruning Example

```
((NOT (RSPP ARP))
 (NOT (CCHSTATEP ACE))
 (NOT (INTEGERP ATG))
 (< ATG '0)
 (NOT (CACHESP SCS))
 (NOT (GOOD-STATE (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                       SCS)))
 (NOT (GOOD-STATE-M (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                         SCS)))
 (NOT (MYSTEP-GOOD (EVENT ETD 'REQE ESD 'DIR)
                   (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                        SCS)))
 (NOT (INTEGERP ETD))
 (< ETD '0)
 (NOT (INTEGERP ESD))
 (< ESD '0)
 (NOT (LOOKUP ETD SCS))
 (NOT (LOOKUP ESD SCS))
 (NOT (LOOKUP I SCS))
 (NOT (INTEGERP I))
 (< I '0)
 (NOT (EQUAL (AGENT-TXN (LOOKUP I SCS))
             'REQE))
 (EQUAL (AGENT-RSP (LOOKUP I SCS)) 'GNTE)
 (NOT (EQUAL (AGENT-TXN (LOOKUP ESD SCS))
             'REQE))
 (AGENT-TAG (LOOKUP ESD SCS))
 (< (DIST-TO-HEAD
      I
      (SYS (AGENT 'NIL ARP 'REQE ACE ATG)
           (UPDATE ESD
                   (AGENT (AGENT-REQ (LOOKUP ESD (CLEAR-ALL SCS)))
                          'GNTE
                          (AGENT-TXN (LOOKUP ESD (CLEAR-ALL SCS)))
                          (AGENT-CCHSTATE (LOOKUP ESD (CLEAR-ALL SCS)))
                          (AGENT-TAG (LOOKUP ESD (CLEAR-ALL SCS))))
                   (CLEAR-ALL SCS))))
    (DIST-TO-HEAD I
                  (SYS (AGENT 'NIL ARP 'NONE ACE ATG)
                       SCS)))
 (EQUAL
    (DIST-TO-HEAD
      I
```

Don’t read this

Well, that’s exactly what pruning does.

Pruning Example (cont)

```
(IMPLIES (AND (NATP ESD)
              (CACHESP SCS)
              (EQUAL (AGENT-TXN (LOOKUP ESD SCS))
                     'REQE))
         (AGENT-TAG (LOOKUP ESD SCS)))
```

read this

We can just throw away everything else. Now we have something that is easy to read and think about. Even better, pruning often will remove terms that can throw ACL2 off track and transform a failing subgoal into one that is proved automatically.

How it works

- Install a trusted clause processor using an override hint combined with a backtrack hint
- Pruning occurs when a subgoal is pushed for induction
- Cache of previous results checked to save redundant effort

Pruning Technique 1

- Naïve algorithm: throw away terms in clause until refutation found, keep any terms whose removal leads to refutation
- Works fairly well, but not perfect
  - Requires test vectors in proportion to number of terms
  - Depending on order of terms removed, may not find smallest pruning
  - Failure to refute non-theorem can lead to pruning needed term as well as adding multiple unnecessary terms

So the easiest way to prune a clause is to throw away terms until you find a refutation. This works fairly well but it's not perfect. For one, each time you remove a term, you need to run a fair number of test vectors to ensure that it wasn't an important term. Also,

Depending on order of terms removed, may not find smallest pruning

For example, imagine the first and last terms in a clause can form a theorem, but a large number of terms in between them can also be used to form a theorem. If we visit the terms in order, the first one will be thrown away since it doesn't immediately result in a non-theorem. Then we will end up keeping the ones in the middle rather than just keeping the first and last term.

Finally, if we use too few test vectors and happen to throw away a term that is actually needed, we may still be able to find refutations due to the missing term as we are testing later terms. This will result in us keeping extra terms we don't need in addition to throwing away one that we did need. So a bad pruning for one term has an effect for the terms that follow it.

Pruning Technique 2

- Observation: many theorems have only a small number of terms
- Idea: look for combinations of terms which always hold
- For each test vector, compute truth value for each term independently
- For all tuples of size n, determine if disjunction is true for all test vectors
- For clause with t terms, requires t^n time/space, but it is practical for small n (I used n < 4)

So in order to get a little better pruning results, I added another algorithm which I try before falling back to the naïve algorithm. This requires more time and space but is practical when the pruning results in theorems with a small number of terms. I found that to often be the case.

Case Study

- Simple coherence protocol (similar to German)
- Additional FIFO to enforce fairness
- 600 LOC for model
- Safety and liveness proofs
- Required about 15 invariants

So the example I was working with while developing this technique was a simple coherence protocol. If you are familiar with the German protocol, it is similar to that. However, it is slightly more complex in one way. I included a FIFO to enforce fairness when the directory is selecting the next request to service. This was necessary in order to be able to prove liveness of the protocol. Before adding the FIFO I was actually able to prove safety without any lemmas, which I thought was quite an impressive feat for ACL2. However, with the additional FIFO there were just too many opportunities to select the wrong induction or get tripped up some other way.

Handling Types

- To increase the effectiveness of testing, testing is biased to only generate ‘well typed’ test vectors
- Before pruning, we remove any type hypotheses since they will never fail
- After we finish pruning, we add back in any type hypotheses for variables mentioned in the clause

This is just one further optimization to ensure our testing is effective and that our pruning results are not missing any required hypotheses.

Results

Without pruning (Enable only hint)

- ACL2 gives up
- 725 pushed first round
- 35.2 avg terms per clause
- tried and failed on five additional subgoals
- (not sure if any can be proved automatically)
- Time: 191.08 seconds (prove: 188.49, print: 2.58, other: 0.01)
- Prover steps counted: 16347149

Prune then enable with cache (~2x speedup vs no cache):

- Proof passes
- 8.5 avg terms per clause
- 318 prunings
- 249 cached
- 15 total inductions after subsumption
- Time: 480.28 seconds (prove: 479.87, print: 0.40, other: 0.01)
- Prover steps counted: 4156823

So this proof was structured in two phases. First, the initial goal was submitted with only a few functions enabled to reduce case splitting. Then I added a hint to enable the remaining functions when subgoals are pushed for induction. In the case where pruning was enabled, I enabled these functions after pruning.

In the proof without pruning, over 700 subgoals were pushed in the first round with an average of over 35 terms per clause. The proof fails on the first pushed induction, and I manually tried the next 5 after that and they all failed as well. The overall time to get to the first failure was 191 seconds.

With pruning enabled, we see the terms per clause drop to 8.5. That is a big improvement and makes looking at failed subgoals a lot more pleasant. You can see there is a fair amount of redundancy in the proof, while pruning 318 subgoals, 249 of those hit in the pruning cache. In the end, ACL2 ends up doing 15 inductions and the proof passes. You can imagine each of those 15 inductions may have required a used supplied lemma without pruning. The proof takes 480 seconds, which is a couple times slower than the failing proof without pruning. I think it could be sped up a bit by improving the cache implementation, but this was fast enough that I didn't bother.

Lemmas

- 1 defstructure type lemma
- 1 expand lemma
- 2 :induct lemmas
- 1 bad pruning lemma
  - Bad pruning appears to be a valid theorem (accidently true?), just not easily provable

So unfortunately, this proof still required a few lemmas. Two of these I would categorize as 'misc' lemmas. One was actually introduced while trying to prove some guards while building the model, the other was to get ACL2 to expand a function wrapped around a cons that it wasn't opening for some reason. The next two lemmas were cases where ACL2 simply chose the wrong induction scheme. These actually could be eliminated if ACL2 were able to backtrack when induction fails and try another scheme. The final lemma was due to a case where the pruning heuristic pruned the wrong terms and returned a non-theorem. Ideally you could also backtrack when there are several promising pruning options as well.

Related Work

- ACL2(s) random testing
- J Moore's coherence proof (433 LOC, 29 lemmas) vs this proof (600 LOC, 5 lemmas)

So I have not done an extensive literature review but here are two related items I know about.

The first is the excellent refutation support already built in to ACL2 Sedan. If you have not tried it yet I highly recommend it. If you have not tried it, it basically uses random testing to find counterexamples to your input conjecture and any subgoals in your proof. Ideally it would be nice if the pruning technique I described were integrated into a more general facility like that. As is, I had to hand code my refutation procedures specifically for this model, but there is no particular reason a more general tool could not be developed.

The second is a paper J wrote about a coherence protocol he proved in ACL2. In some ways it is more comprehensive, because it reasons about memory consistency rather than just coherence. However, the protocol itself is somewhat more atomic than the one I have modeled and does not include a liveness proof. Overall, it has about 30% fewer lines of code and more than 5 times as many lemmas.

Future Work

- Create a general pruning facility for ACL2
- Improve cache performance
- Increase backtracking / proof search
- Invariant creation

So these items are mostly straightforward. I will just say a bit about invariant creation, since I had to create a somewhat complex invariant for this proof. After you have gotten all the bugs out of your model, if you have disabled generalization and your subgoals are mostly reliable, the remaining proof failures you will see are things that needed to be added to your invariant. I think if we get to the point where proofs are highly automated, you could automatically extract many of these invariants from failed subgoals.

Lessons Learned

- Turn off generalization (or use testing to backtrack)
- Defstructure + Destructor Elimination works nicely
- :otf-flg ‘ftw

A few lessons I learned while doing this proof. I’m sure many of you are already familiar with these.

First, generalization fails often. I found it best to just disable it. It just creates more noise than it is worth. I think some of you might know that ACL2(s) uses testing to backtrack when generalization fails. That is pretty cool. I was using raw mode for my clause processor so I didn’t get to use that but that would be the best of both worlds, I think

Second, using a records book along with destructor elimination is very powerful. That gives you a type of generalization that is fairly safe. I was kind of amazed how many subgoals were proved by induction by after using this technique.

Finally, for the type of theorems I was proving, I found otf-flg to be very useful. I was a bit afraid at first because of the name that I was doing something that would not work out. But it turned out to be the right choice.

More details on records + destructor elim:

As for the records book, the one I'm using is defstructure, although there at least two others I know of called defaggregate and defdata. Not sure if you have used any of those before, but they basically let you define data structures with

accessors/destructors so that you don't have to use car cadr caddr etc everywhere. Instead you can define your own data structures with fields and access them using reasonable names like (caches s) (dir s) etc.

Anyway, if you are familiar with how destructor elimination works with car/cdr, then you can imagine something similar for these accessor functions. For example, if you have a record s (of your type, lets call it sys) and an accessor (caches s) and a elim rule for your datatype, ACL2 can replace that term with a fresh variable C as long as it can establish (sysp s).

This is important for induction, as leaving that extra accessor around can mess up things. Anyway, the advantages are similar to car/cdr elim but for subtle reasons some of the records books (defdata I know for sure) do not currently support dtor elim. defstructure does and I had a lot more luck automating my proofs once I got that working.

Conclusions

- Testing is a very powerful tool
- Can be used to automate proofs by pruning subgoals
- The future of theorem proving is highly automated!