Backtracking and Induction in ACL2 [*]

John Erickson
University of Texas at Austin
jderick@cs.utexas.edu

ABSTRACT

This paper presents an extension to ACL2 that allows backtracking to occur when a proof fails. Using this extension, two techniques are implemented for proving theorems. The first of these allows ACL2 to find alternate substitutions for unmeasured variables during induction. The second allows ACL2 to try alternate generalizations when one fails. These techniques combine to allow ACL2 to prove theorems that it could not prove before.

1. INTRODUCTION

Often, ACL2[6] has to choose between several promising alternatives during the course of a proof. For example, a given theorem may suggest three possible induction schemes. ACL2 will choose one and proceed. However, if the proof fails, ACL2 has no mechanism for returning to the point at which the choice was made and attempting an alternate induction. In this paper, we describe an extension to ACL2 that allows such backtracking to occur. This extension enables us to experiment with many new theorem proving heuristics.

We describe two such heuristics and show how they can be used to automatically prove theorems that ACL2 could not prove automatically before. The first of these is an induction variable matching algorithm that allows ACL2 to automatically generate new induction schemes for theorems about functions with unmeasured variables. This algorithm in based on a paper by Kapur and Subramaniam [5] and allows ACL2 to prove theorems such as `(equal (rot (len x) (append x y)) (append y x))`, where an induction scheme must be discovered that substitutes `(append y (list (car x)))` for `y`, when no such scheme is suggested by the functions involved.

The second heuristic is related to cross fertilization. There are times when ACL2 will choose to cross fertilize and generalize when it would be better to skip cross fertilization and generalize directly. With our extension, both possibilities can be tried.

[*]This work supported by DARPA CyberTrust project number 0429591

2. INDUCTION VARIABLE MATCHING

In ACL2, when an induction scheme contains unmeasured variables, the scheme can be modified to yield any substitution for those variables. Since only unmeasured variables are modified, the measure will be unchanged, and the scheme will remain sound.

Although this allows a great deal of flexibility in choosing an induction scheme, it can be difficult to find the right substitutions for a given variable. The technique we use is based on a paper by Kapur and Subramaniam [5]. The main idea is to replace the unmeasured variables with new constrained functions in the induction hypothesis, and then attempt to find definitions for them by attempting to match the induction conclusion ("IC") and the induction hypothesis ("IH") after simplification. Differences are eliminated by removing recursive functions using case splits. After definitions are found for the constrained functions, they are substituted back into the original induction step. If any differences remain, lemmas are speculated to remove them.

For an example of when induction variable matching can be helpful, consider the theorem[1]

```
(implies (and (true-listp x) (true-listp y))
         (equal (rot2 x (append x y))
                (append y x)))
```

where `rot` is defined as:

```
(defun rot (n x)
  (if (zp n)
      x
    (rot (1- n) (append (cdr x) (list (car x))))))
```

ACL2 will attempt to induct on `(cdr x)`, leaving `y` unchanged. This gives

[1]Here we show the full theorem with true-listp hypotheses. In the rest of the paper, we will present this example as if these hypotheses were not needed. However, our system must track these in order to correctly prove this theorem and does so.

```
(equal
 (rot (len (cdr x)) (append (cdr x) y))
 (append y (cdr x)))
```

for the induction hypothesis. After stepping and simplifying the induction conclusion, we get:

```
(equal
 (rot (len (cdr x))
      (append (append (cdr x) y) (list (car x))))
 (append y x))
```

The IH,

```
(equal
 (rot (len (cdr x)) (append (cdr x) y))
 (append y (cdr x)))
```

cannot be applied. But note that if in this IH we further replaced y with `(append y (list (car x)))` and we knew that append was associative, the new IH would apply and finish the proof. The challenge is finding this instantiation of `y`.

To do this, we start by replacing `y` with the constrained function `(F x y)` in the induction hypothesis to get the following induction hypothesis:

```
(equal
 (rot (len (cdr x)) (append (cdr x) (F x y)))
 (append (F x y) (cdr x)))
```

Attempting to match the LHS of the IH and simplified IC reveals the following differences:

```
IH (append (cdr x) (F x y))
IC (append (append (cdr x) y) (list (car x)))
```

We try to match these by assuming the base case for the inner most `append`, namely `(endp (cdr x))`, and simplifying. This gives us:

```
IH (F x y)
IC (append y (list (car x)))
```

which gives us a definition for `F`. Since this definition was generated using a special case, we substitute it back into the constrained IH,

```
(equal
 (rot (len (cdr x))
      (append (cdr x) (append y (list (car x)))))
 (append (append y (list (car x))) (cdr x)))
```

and then compare this to the simplified IC,

```
(equal
 (rot (len (cdr x))
      (append (append (cdr x) y) (list (car x))))
 (append y x))
```

to see what differences remain.

On the LHS, this gives us:

```
IH (append (cdr x) (append y (list (car x))))
IC (append (append (cdr x) y) (list (car x)))
```

On the LHS, this difference can be proved as a lemma. If we also generalize by replacing common subterms with new variables, we obtain the associativity of `append`:

```
(equal (append (append x y) z) (append x (append y z)))
```

On the RHS, we have these differences:

```
IH (append (append y (list (car x))) (cdr x))
IC (append y x)
```

This lemma is a special case of associativity, and can be recognized as redundant.

3. MULTIPLE GENERALIZATIONS

It is well known that many theorems must be generalized before they can be proved. For example, `(equal (rev1 x nil) (rv x))` is typically generalized to `(equal (rev1 x a) (append (rv x) a))`. Although ACL2 already has the capability to generalize theorems, often it will choose a bad generalization. If this happens, ACL2 will not try another generalization; it will simply fail. Our extension allows alternate generalizations to be attempted.

The system generates two alternatives during every induction. First, it will try to cross fertilize and then generalize any remaining goals before another induction. If that fails, it will throw away the IH and generalize the remaining goals without cross fertilization. Below we give several examples of when this alternate generalization will succeed.

3.1 Reverse Example

As an example, consider the theorem `(equal (rv1 x nil) (rv x))`, where rv and rv1 are defined as:

```
(defun rv (x)
  (if (endp x)
      nil
    (append (rv (cdr x)) (list (car x)))))

(defun rv1 (x a)
  (if (endp x)
      a
    (rv1 (cdr x) (cons (car x) a))))
```

ACL2 will induct on `(cdr x)` to prove this theorem. After simplification and destructor elimination, the induction step will be:

```
(implies (equal (rv1 x2 nil) (rv x2))
         (equal (rv1 x2 (list x1))
                (append (rv x2) (list x1))))
```

ACL2's normal behavior is to cross fertilize after this step, yielding:

```
(equal (rv1 x2 (list x1))
       (append (rv1 x2 nil) (list x1)))
```

Cross fertilization has reintroduced the constant `nil` into the accumulator of `rv1`. This will make proving the above goal difficult. If instead, we throw away the IH and generalize by replacing `(list x1)` with `x3`, we get:

```
(equal (rv1 x2 x3)
       (append (rv x2) x3))
```

which can be proved by ACL2.

3.2 Rotate Example

Consider the theorem `(equal (rot (len x) x) x)`. After simplification and destructor elimination, the induction step will be:

```
(implies
 (and (equal (rot (len x2) x2) x2)
      (true-listp x2))
 (equal (rot (len x2) (append x2 (list x1)))
        (cons x1 x2)))
```

after cross fertilization, we get:

```
(implies
 (true-listp x2)
 (equal (rot (len x2) (append x2 (list x1)))
        (cons x1 (rot (len x2) x2))))
```

which ACL2 further generalizes to the non-theorem:

```
(implies (and (integerp i)
              (<= 0 i)
              (true-listp x2))
         (equal (rot i (append x2 (list x1)))
                (cons x1 (rot i x2))))
```

If instead, we skip cross fertilization and throw away the induction hypothesis, we get:

```
(defthm car-ap-cons
  (equal (car (append (cons a b) c))
         a))

(defthm cdr-ap-cons
  (equal (cdr (append (cons a nil) c))
         c))

(defthm append-cons
  (consp (binary-append (cons x3 nil) z))
  :rule-classes :type-prescription)

(defthm cons-ap
  (implies (syntaxp (not (equal x ''nil)))
           (equal (cons a x)
                  (append (cons a nil) x))))
```

Figure 1: cons to append normalization rules

```
(implies
 (true-listp x2)
 (equal (rot (len x2) (append x2 (list x1)))
        (cons x1 x2)))
```

In this case, there are no common subterms across the equality, so generalization fails. However, notice that the element `x1` occurs at the end of the list in the accumulator on the LHS and at the beginning of the list on the RHS. If we use the rules in Figure 1 to normalize lists, the goal above becomes:

```
(implies
 (true-listp x2)
 (equal (rot (len x2) (append x2 (list x1)))
        (append (list x1) x2)))
```

Now we can generalize, because the term `(list x1)` appears on both sides. This gives

```
(implies (true-listp x2)
         (equal (rot (len x2) (append x2 x3))
                (append x3 x2)))
```

This theorem we proved earlier using unmeasured variable matching.

4. IMPLEMENTATION

Our implementation uses ACL2's simplification and generalization routines along with our own version of induction. Our induction routines replace unmeasured variables in the induction hypothesis with constrained functions for which we will later find definitions. Instead of using ACL2's top level prover, we have our own control flow that allows induction to be entered and exited recursively. Below we present pseudocode for our implementation. The top level function, shown in Figure 2 below, is called *bprove*. It takes a term and attempts to prove it, returning either SUCCESS or FAILURE.

```
bool bprove(term x)
{
  l := bash(x)

  for each permutation p of l
    success := prove-perm(p)
    if success
      return SUCCESS
    else
      continue

  return FAILURE
}
```

Figure 2: The top level function for the backtracking prover.

```
// l is a list of clauses
bool prove-perm(list l)
{
  while l is non-empty
    c, l := remove-clause(l)
    if there are any constrained functions in c
      c, success, bind := remove-constraints(c)
      if !success
        return FAILURE
      l := apply-subst(bind, l)

    refuted := refute(c)
    if refuted
      return FAILURE

    success := binduct(c)
    if !success
      c := generalize(c)
      success := binduct(c)
      if !success
        return FAILURE

  return SUCCESS
}
```

Figure 3: Prove a list of clauses

```
(clause, bool, list) remove-constraints(clause c, list bind)
{
  for each literal l in c
    if l is of the form (not (equal lhs rhs)), attempt
    to match each side of the equality against all subterms of the
    other literals in the clause, replacing lhs[rhs] with rhs[lhs]
    wherever applicable

    if an equality successfully matches, remove it from the clause and
    call remove-constraints again on the remaining literals, along
    with any bindings aquired during the match

    if l is of the form (not l'), attempt to match l'
    against the other literals in the clause

    if the match is successful, then we have found two literals that
    are negations of each other.  We return SUCCESS along with the
    substitutions returned from match and the empty clause.

  if any constrained functions remain, return FAILURE
  else return SUCCESS along with the modified clause and any bindings
}
```

Figure 4: Remove constraints

```
// returns a list of bindings for any constrained functions if
// successful
(bool, list) match(term a, term b)
{
  if neither a nor b contain any constraints
    call bprove on ``a = b''

  if a contains no constraints
    switch a and b

  if the top symbol of a is a constrained function
    bind the constrained function to b and return SUCCESS

  if a and b have the same top symbol, decompose them and attempt to
  match corresponding subterms

  if that fails, let h be the simplifying assumptions attained by
  assuming the base case for the innermost recursive function in a,
  and return match(a', b'), where a' and b' are simplifications of a
  and b under hypotheses h

}
```

Figure 5: Match two terms modulo constrained functions

Our prover starts with a call into the simplifier using the bash book developed by Matt Kaufmann. This simplifier returns a list of clauses. We must prove all clauses in order to prove our goal theorem. Since there may be constrained functions in the clauses that will be bound to concrete functions as we proceed, the order that we prove the clauses is important. This is because, while proving a clause, we may discover bindings for constrained functions in that clause. These bindings will be used to remove any instances of the same constrained function in later clauses. Furthermore, different clauses may find differing bindings for the same constrained function. Therefore, it is necessary to try to prove all permutations of a given clause list. Figure 3 shows the pseudocode for the function *prove-perm*, which is used to prove such a permutation. For performance reasons, our implementation does not actually compute the entire set of permutations at once. Instead, we generate one at a time. This allows us to avoid unnecessary work if we find a proof early.

For each clause, we first remove any constraints. Any bindings acquired by removing constraints are applied to the remaining clauses. Next, we attempt to refute the clause, by generating a number of finite cases of the theorem and sending them through the simplifier. If no counterexamples were found, we induct with cross fertilization and generalization. If that fails, we throw away any induction hypothesis and generalize before attempting a second induction. These calls to *binduct* use our own induction mechanism so that we can annotate the induction hypothesis with constrained functions if there are any unmeasured induction variables. The goals generated by this induction are then fed back into our prover via the function *bprove*. The proof search terminates because it is bounded by a maximum number of nested inductions. This limit is usually set to 3 but can be set to any number.

4.1 Removing Constraints

Removing constraints is done using the function *remove-constraints* from Figure 4. In order to satisfy the constraints for a given clause, we visit all pairs of literals in the clause. For a given pair, if one literal is negated and the other is not, we attempt to match them. Also, if one is a negated equality, we will attempt to match the lhs[rhs] or the equality against all subterms of the other. If the match is successful, we substitute terms using the equality, remove the equality from the clause, and then attempt to prove the clause without the equality.

There are two techniques we use to match two terms, as shown in Figure 5. First, if two terms have the same top function symbol, we will decompose the terms and attempt to match their subterms. Second, if two terms do not share the same top symbol, we will simplify the terms by assuming the base case of the inner most recursive call in the first term. By repeating these two procedures, we guarantee that eventually all recursive functions will be removed from the first term. In such a case, one way unification can be used to determine if a definition has been found for any constrained functions remaining. Subterms that contain no constrained functions will be sent back to the prover to determine if they are equal.

4.2 Refuting Conjectures

Matching creates many subgoals that are easily disproved. We use a simple technique to refute such goals that can deal with most ACL2 formulas. Doing so avoids sending the prover down many dead end paths. For formulas where no recursive functions are present, we send the formula through the simplifier. If the simplifier fails to prove the formula, we assume the formula false. For functions with recursive calls, we find any recursive function call in the formula and open it up, creating a number of new formulas with a case split. We then recur on these formulas. In theory, because these functions must terminate, for any invalid conjecture there exists a finite depth at which this procedure will find a refutation. However, we limit the depth of the search, typically to a maximum of five nested case splits. We have found this technique to be effective for eliminating obviously false conjectures.

5. RELATED WORK

Rippling [2] and proof planning [1] are two of the more well known techniques for automating induction. Other more recent techniques for automatically proving theorems by induction include higher order rippling with proof critics [3] and cover sets with decision procedures [4].

6. CONCLUSIONS

ACL2 has powerful heuristics which can often prove theorems automatically. However, there are times when several reasonable alternatives exist. Allowing ACL2 to try more than one alternative and backtrack in the case of failure results in more theorems proved. We presented two such scenarios. First, when an induction scheme contains unmeasured variables, there may be many different viable substitutions for those variables. Second, after induction, it may sometimes be useful to throw away the induction hypothesis and generalize before continuing. Our implementation allows the possibility of extending ACL2 with even more search capabilities. As computers become faster, especially as multi-core processors become more widespread, these search capabilities offer the possibility to take advantage of this computing power for the purpose of proving theorems.

7. REFERENCES

[1] A. Bundy. The use of explicit plans to guide inductive proofs. In *Conference on Automated Deduction*, pages 111–120, 1988.

[2] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[3] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.

[4] D. Kapur. Rewriting, decision procedures and lemma speculation for automated hardware verification. In *Theorem Proving in Higher Order Logics*, pages 171–182, 1997.

[5] D. Kapur and M. Subramaniam. Lemma discovery in automating induction. *Lecture Notes in Computer Science*, 1104:538–??, 1996.

[6] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer

Academic Publishers, 2000.