CSU290 Lecture Notes   Lecture 28   1 Dec 2008

Peter Dillinger

More proofs in ACL2

Today we will guide ACL2 in some proofs regarding set theory, a branch of mathematics concerned with collections of values. Here are two definitions to start off with:

```
(defun mem (e x)
  (if (endp x)
    nil
    (if (equal e (car x))
      t
      (mem e (cdr x)))))

(defun subset (x y)
  (if (endp x)
    t
    (and (mem (car x) y)
         (subset (cdr x) y))))
```

MEM of course checks whether e, an "element," is a member of a list x. If we represent a set of objects as just a list of all the elements, then MEM can be thought of as checking set membership.

SUBSET checks if x is a subset of y--if all the elements of x are elements of y. Note that any set is a subset of itself: any set contains all its own elements. Let’s try to prove that:

```
(defthm subset-reflexive
  (subset x x))
```

but it fails. The key checkpoint under induction is

```
(IMPLIES (SUBSET X2 X2)
         (SUBSET X2 (CONS X1 X2)))
```

If we ask ACL2 to prove that, it fails. Can we make this lemma more general? If so, it could be easier to prove, by setting up the correct induction, and would be more reusable as a rewrite rule.

Conceptually, it’s not important that the second parameter to SUBSET be the same as the first for this to be true:

```
(defthm subset-cons
  (implies (subset x y)
           (subset x (cons e y))))
```

That is proven automatically and allows ACL2 to prove subset-reflexive.

Another property we want to hold of SUBSET is that it is transitive:

```
(defthm subset-transitive
  (implies (and (subset x y)
                (subset y z))
           (subset x z)))
```

That fails, with the following key checkpoint under induction:

```
(IMPLIES (AND (NOT (MEM X1 Z))
              (MEM X1 Y)
              (SUBSET X2 Y))
         (NOT (SUBSET Y Z)))
```

ACL2 does not prove that automatically, and it turns out a very simple generalization enables the proof to go through: the hypothesis (SUBSET X2 Y) is completely irrelevant. X2 is not even mentioned elsewhere in the conjecture, but if that hypothesis is included, ACL2 choses the wrong induction scheme. Here is a version without that hypothesis and with some variables renamed, etc.:

```
(defthm not-subset
  (implies (and (mem e x)
                (not (mem e y)))
           (not (subset x y))))
```

And that allows the transitivity proof to go through.

Another operation performed on sets is intersection, which is to find all elements that are members/elements of both of two given sets:

```
(defun int (x y)
  (if (endp x)
    nil
    (if (mem (car x) y)
      (cons (car x) (int (cdr x) y))
      (int (cdr x) y))))
```

One thing that should be true is that the intersection should be a subset of either parameter. Both of these proofs are automatic for ACL2:

```
(defthm int-subset
  (subset (int x y) x))

(defthm int-subset2
  (subset (int x y) y))
```

But for something more sohpisticated, observe that the order of the arguments to an intersection operation should not matter. The same set should be returned regardless of order. We conjecture

```
(defthm int-symmetric
  (equal (int x y)
         (int y x)))
```

And that fails to prove. In this case, it fails because it is not a theorem! Here’s a counterexample:

```
  (int ’(1 2 3) ’(4 3 2))  =  ’(2 3)
  (int ’(4 3 2) ’(1 2 3))  =  ’(3 2)
```

These are not equal in the ACL2 sense, but they represent the same set. Conceptually, elements of a set don’t have an order, but they do when representing them as lists.

Can we come up with a notion of set equality and use that? Of course. Two ACL2 objects represent equivalent sets if they contain the same elements. The easiest way to define that in terms of what we have defined is

```
(defun set= (x y)
  (and (subset x y)
       (subset y x)))
```

--- BEGIN Miscellaneous mathematics that will not be specifically tested. ---

We want to be sure this is what mathematicians call an Equivalence Relation, which divides the universe of values in to groups called Equivalence Classes. Within each class, all the values are "equivalent" according to the relation, and no values in different classes are "equivalent."

Mathematicians tell us there are three properties that must hold for something to be an equivalence relation: reflexivity, symmetry, and transitivity.

```
(defthm set=-reflexive
  (set= x x))

(defthm set=-symmetric
  (implies (set= x y)
           (set= y x)))

(defthm set=-transitive
  (implies (and (set= x y)
                (set= y z))
           (set= x z)))
```

ACL2 proves these automatically, and once we have done so, we can tell ACL2 that it can use SET= in special ways as an equivalence relation:

```
(defequiv set=)
```

--- END Miscellaneous mathematics that will not be specifically tested. ---

So what we want to prove about intersection is actually

```
(defthm int-commutative
  (set= (int x y)
        (int y x)))
```

As we might expect, this fails. If we follow our usual method of just looking at checkpoints under induction, generalizing, and proving those as lemmas, we do a lot of work. Notice, however, that ACL2 was able to do some work before reverting to induction:

```
  By the simple :definition SET= we reduce the conjecture to the following
  two conjectures.

  Subgoal 2
  (SUBSET (INT X Y) (INT Y X)).

  Name the formula above *1.

  Subgoal 1
  (SUBSET (INT Y X) (INT X Y)).
  ^^^ Checkpoint Subgoal 1 ^^^

  Normally we would attempt to prove Subgoal 1 by induction.  However,
  we prefer in this instance to focus on the original input conjecture
  rather than this simplified special case.  We therefore abandon our
  previous work on this conjecture and reassign the name *1 to the original
  conjecture. ...
```

Now there is something interesting about these two subgoals. Suppose we are able to prove just one of them as a lemma. Then the other is just an instantiation of with X replacing Y and Y replacing X! Thus, if we prove

```
(defthm int-comm-lemma
  (subset (int x y)
          (int y x)))
```

then ACL2 will be able to prove INT-COMMUTATIVE.

The proof fails and there are three checkpoints under induction. Two of them are quite similar:

```
Subgoal *1/3'4'
(IMPLIES (AND (NOT (MEM X1 Y))
              (SUBSET (INT X2 Y) (INT Y X2)))
         (SUBSET (INT X2 Y)
                 (INT Y (CONS X1 X2))))

Subgoal *1/2.1''
(IMPLIES (AND (MEM X1 Y)
              (SUBSET (INT X2 Y) (INT Y X2)))
         (SUBSET (INT X2 Y)
                 (INT Y (CONS X1 X2))))
```

In this case, I can save myself some work if I really think about what I need to prove for ACL2 to be able to prove these. In particular, I know that SUBSET is transitive. Thus, if I prove that (INT Y X2) is a subset of (INT Y (CONS X1 X2)), then both of these will be proven by the transitivity of SUBSET. I have renamed some variables for elegance:

```
(defthm subset-int-cons
  (subset (int x y)
          (int x (cons e y))))
```

That is proven automatically, and it just leaves the other checkpoint under induction:

```
Subgoal *1/2.2''
(IMPLIES (AND (MEM X1 Y)
              (SUBSET (INT X2 Y) (INT Y X2)))
         (MEM X1 (INT Y (CONS X1 X2))))
```

If you think about it, the second hypothesis is irrelevant. This gives us the lemma

```
(defthm mem-int-cons
  (implies (mem e x)
           (mem e (int x (cons e y)))))
```

which is proven automatically and enables the proof of INT-COMM-LEMMA and INT-COMMUTATIVE.

But we could be more general with that lemma, and it would be a more useful rule:

```
(defthm mem-int
  (implies (and (mem e x)
                (mem e y))
           (mem e (int x y))))
```

That is also automatic and enables proof of INT-COMM-LEMMA and INT-COMMUTATIVE.